Online Performance-Improvement Algorithms

Prasad Chalasani

August 1994
CMU-CS-94-179

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Avrim Blum, Chair
Daniel Sleator
Michael Erdmann
Prabhakar Raghavan, IBM T.J. Watson Research Center

This research was sponsored by the National Science Foundation and the Young Investigator Program under Grant No. CCR-9357793. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government or NSF.

Keywords: Learning, Robot Navigation, Performance Improvement, Online Algorithms, Competitive Analysis, Task Systems, Paging, Switching Concepts

Carnegie Mellon

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

Competitive Online Learning Algorithms for Navigation, Paging, and Predicting Switching Functions

PRASAD CHALASANI

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

Avrim Blum — 8/18/94
THESIS COMMITTEE CHAIR — DATE

[signature] — 8/29/94
DEPARTMENT HEAD — DATE

APPROVED:

R. RM — 8/29/94
DEAN — DATE

Abstract

The results presented in this thesis contribute to two areas of computer science: machine learning and on-line algorithms. One limitation of current theoretical approaches to learning is that most of them involve function approximation, i.e., the learner improves the accuracy of its approximation to an unknown function as it sees more samples of the function. This thesis presents algorithms for improving performance at other tasks, such as navigation and and paging. Specifically, these algorithms are online algorithms whose competitiveness improves when repeatedly applied to the same problem. The design of such algorithms is a new challenge in the area of online algorithms: the algorithms must not only be competitive on each application, but must also acquire useful information to improve their future competitiveness. We present a general framework based on task systems for studying such algorithms.

One result in the thesis is an online navigation algorithm for a robot traveling back and forth between two points s and t (distance n apart) in a scene filled with unknown axis-parallel rectangular obstacles. For each $i \leq n$, the ith trip of the robot is guaranteed to be within a $O(\sqrt{n/i})$ factor of the shortest s-t path length L, and we show that up to constant factors this is the best a deterministic algorithm can do. This algorithm is based on a smooth search-quality tradeoff: we design an algorithm that given any $k \leq n$, searches a distance $O(L\sqrt{n/k})$ and finds an s-t path of length $O(L\sqrt{n/k})$. A key insight is that this tradeoff can be achieved by optimally traversing a certain tree structure based on the obstacles.

For a version of the paging problem where the pager is aware only of page faults, we give an algorithm whose average competitiveness improves when the same page request sequence is repeated several times.

The thesis also applies competitive analysis to design an online algorithm for a natural extension of the standard function prediction problem, where the function labeling the examples switches unpredictably between two unknown functions.

Acknowledgements

I was very fortunate to have been working with Avrim Blum, my thesis advisor. I benefited greatly from his keen sense of what research directions are good to pursue, and I was inspired by his commitment to high quality work. All of the work described in this thesis is joint work with Avrim. Our collaboration was especially fun because Avrim always created opportunities for me to do original work, and this has helped boost my confidence tremendously. He has always been encouraging and enthusiastic, and generous with his time. During these final days of my PhD he has meticulously gone over drafts of my thesis. For all of this I am very grateful to Avrim, and I want to thank him for giving me a start in my research career.

I would like to thank my committee members Prabhakar Raghavan and Daniel Sleator for comments on an earlier draft of this thesis. Thanks to them and also to Michael Erdmann for interesting discussions, for their encouragement, and for serving on my thesis committee.

In my initial years at CMU I have had several stimulating discussions with Tom Mitchell and Herb Simon, for which I am very thankful. I also want to thank Manuel Blum, Alan Frieze and Stephen Suen for the opportunity to collaborate with them, and for their enthusiasm in listening to my ideas.

I want to thank my officemates Greg Morrisett and Andrzej Filinski, and my former officemate Mike Gleicher, for always being ready to help. I am also indebted to them and many other friends for making my graduate student life interesting and fun.

I would like to express my gratitude to the many people in the department who have helped me over the years. Special thanks to Sharon Burks for all her advice and encouragement, and to Catherine Copetas, who helped me on numerous occasions and brightened up the atmosphere with her friendly ribbing.

I am most of all thankful to my parents and my brother for their unfailing support, patience and encouragement through the years of my PhD.

Contents

1 Introduction 1

2 Performance-Improvement Algorithms 9

3 Improving Performance in Navigation 15

3.1 Introduction 15

3.2 The model, and some preliminaries 17

3.3 A Lower Bound for k Trips 18

3.4 The Optimal k-trip Cumulative Algorithm 20

3.5 The Algorithm for Simple Scenes 21

3.5.1 Fences and the One-trip BRS Algorithm 22

3.5.2 A Naive k-trip Approach 23

3.5.3 Groups of Disjoint Fences 26

3.5.4 Fence-trees 29

3.5.5 Traversing the Fence-tree 37

3.6 Extension to Arbitrary Axis-Parallel Rectangular Obstacles 46

3.6.1 τ-Posts 47

3.6.2 τ-Fences 47

3.6.3 The Initial Search Trip 48

3.6.4 Extending FindFenceTree to General Scenes 49

3.6.5 Modifying JumpDownRight 51

3.6.6 Modifying JumpDownLeft 53

3.7 An incremental algorithm 55

3.8 Modification for Point-to-Point Navigation 57
3.9 The Navigation Problem Viewed as a Hidden Task System 57
3.10 Conclusion and Open Problems 59
3.11 Related Work 60

4 Improving Performance in Paging **65**
4.1 Introduction 65
4.2 Preliminaries 68
4.3 The Algorithm 68
4.4 Analysis 70
4.5 The Paging Problem Viewed as a Hidden Task System 73
4.6 Conclusion and Open Problems 74
4.7 Related Work 74

5 Online Learning of Switching Concepts **79**
5.1 Introduction 79
5.2 Notation and Model 80
5 3 The Algorithm and Analysis 82
5.4 Open Problems 86
5.5 Related Work 86

List of Figures

3.1 A full brick pattern 19
3.2 A simple scene. 23
3.3 Colliding Fences 25
3.4 A Repair 25
3.5 Repair is expensive 26
3.6 An efficient construction 27
3.7 Optimal Algorithm 28
3.8 Initial search trip, $k = 1$ 30
3.9 Initial search trip, $k = 2$. 31
3.10 Initial search trip, $k = 3$. 32
3.11 Initial search trip, $k = 6$. 33
3.12 A Fence Tree, simple scene 34
3.13 Procedure **JumpDownLeft**, simple scene 41
3.14 Procedure **JumpDownLeft** over-charges 42
3.15 Procedure **JumpDownRight**, simple scene 43
3.16 Bands. 48
3.17 A Fence Tree 49
3.18 Procedure **JumpDownRight** 52
3.19 Procedure **JumpDownLeft** 56

4.1 A hole path 68
4.2 Algorithm **SleepyPager** 71
4.3 A cell tree 72

Chapter 1

Introduction

The results in this thesis contribute to two areas of computer science: machine learning and online algorithms. The thesis shows how algorithms for certain learning problems can be designed using the framework and techniques of online algorithms.

Performance-Improvement Algorithms

One goal of machine learning research is to develop programs that improve their performance at some task as they gain more experience with the task. For instance one might want to develop a speech recognition system that improves its recognition accuracy as it interacts with speakers. Another example is a weather forecasting system that makes better predictions as it gains experience. Similarly one might want a robot to be able to improve its performance in path planning when walking back and forth between two places in an unknown environment such as mars.

There has been much empirical research on designing programs that improve their performance at various tasks. Examples include the work on speech recognition [56], character recognition [55], gesture recognition [28], autonomous vehicles [71], robotic manipulation [26], robot navigation [24], and adaptive user interfaces [66]. Alongside with this empirical work, there have been attempts at capturing mathematically what it means to "learn", and developing algorithms which can be shown to learn in an appropriate rigorous sense. The hope is that by abstracting away the details of individual tasks, general learning strategies can be developed and applied to particular problems. In addition one hopes to gain a better understanding of what kinds of learning tasks are inherently difficult. The mathematical models introduced in the pioneering work of Angluin [4, 5, 7], Gold [33, 34, 35] and later Valiant [81] have lead to the development of a field that has come to be known as *computational learning theory.*

These formalisms generally only model the task of *function-approximation*, (or function-prediction) where the goal is to develop algorithms that improve their quality of approximation of an unknown function (or automaton) as they see more samples of its input-output behavior. While it is true that a variety of tasks can be viewed as function-approximation tasks, such a view may not always be the most natural one. Thus one would like to develop models and algorithms for performance-improvement at tasks (such as the ones mentioned above) other than just function approximation.

This thesis takes an initial step in this direction. In the function approximation task, the learner uses partial information about the target function, namely its value at some points, in order to predict the value of the function at other points. As it acquires more information, it improves its prediction accuracy. To extend learning models beyond function-prediction tasks, we consider *online optimization problems*, where an algorithm can improve the quality of its solution by acquiring more information about the problem input. Before moving on, we review the basic vocabulary of online algorithms.

Online Algorithms and Competitiveness

An optimization problem requires an online solution when the input is revealed a piece at a time, and the algorithm must make a decision as it sees each piece, before the remaining pieces are revealed. Naturally, the quality of the solution obtained by an online algorithm could be significantly worse than that of the optimal offline solution for the problem (i.e., the optimal solution that would be computed by an algorithm that knows the entire input).

Many real-world optimization problems are online in nature. For instance, paging is an online problem where one would like to incur as few page faults as possible. When faced with a page fault, the pager (i.e., the paging algorithm) must decide which page to evict from fast memory before future page requests are known. Since the pager does not know the future, on a given sequence of page requests the pager might suffer many more page faults than a (hypothetical) optimal pager that knows the entire sequence of requests. Another online optimization problem arises when a robot is trying to go as quickly as possible from point s to point t in a scene filled with unknown obstacles. The problem is online since the robot must decide which way to move when it encounters an obstacle without (full) knowledge of what obstacles may lie ahead. Again, the length of the robot's path from s to t could be much longer than the optimal s–t path that could be computed given a map of the entire scene.

It is customary to call an online algorithm *c-competitive* (c is called the *competitive ratio*) if on any sufficiently long input sequence, its cost (e.g., number of page faults) is bounded by a factor c times the cost of the optimal offline solution to the problem. (The interpretation of the "length" of the input sequence depends on the particular

online problem at hand.)

Improving Competitiveness using Partial Information

For many non-trivial online problems, even the best possible online algorithm has a large competitive ratio. However if the algorithm obtains some partial information about the input (e.g., future page requests, or obstacles that lie ahead in the scene), it may be able to improve its competitiveness. For instance such partial information may be available when the algorithm is repeatedly applied to the same problem instance, and on each application it can only see a portion of the input. This gives rise to the possibility that on each application, the algorithm can accumulate information, and use it to improve its competitiveness on future applications. This thesis considers the design of online algorithms whose performance provably converges quickly to that of the optimal offline algorithm upon repeated application to a problem. (This will become clearer in the particular problems we consider.) In particular we develop such algorithms for some natural problems in robot navigation and paging (described below).

The study of how partial information can be exploited to improve competitiveness in online algorithms is relatively new. Some researchers [3, 40] have analyzed the benefits of *lookahead* (e.g., having access to the next few page requests) on competitiveness. Others have considered paging problems where the request sequence is not completely arbitrary but must satisfy some constraints [21].

Of course partial information does not always help improve competitiveness. Thus a new challenge in the problems we consider is to design an algorithm that on each application not only is competitive, but also acquires *useful* information that helps improve future competitiveness. The design and analysis of such algorithms can be seen as a contribution to the area of online algorithms.

In Chapter 2 we present a general model for studying online algorithms whose performance improves with repetition. This model is based on the task system model introduced by Borodin, Linial and Saks [23]. Chapters 3 and 4 present two performance-improvement algorithms, and we briefly introduce these in the next two subsections.

Problem 1: Improving Performance in Robot Navigation

Chapter 3 describes a performance-improvement algorithm for the following robot navigation scenario: There is a two-dimensional scene containing non-overlapping (and non-touching) axis-parallel rectangular obstacles of unknown sizes and positions. A point robot is required to travel back and forth (avoiding obstacles) quickly between two points s and t in the scene. The robot initially does not know the sizes and

positions of the obstacles; it only has tactile sensing, i.e., it discovers obstacles when it touches them. The robot's strategy must therefore be online: the robot must decide which way to move when it hits an obstacle, without (full) knowledge of what obstacles may lie ahead.

A natural question is whether there is a strategy for the robot to walk from s to t in such a way that the ratio (the "competitive ratio") of its trip length to the shortest obstacle-free s-t path length is not too large. Papadimitriou and Yannakakis [70] showed that in any scene where the Euclidean s-t distance is n units (where the width of the thinnest obstacle is taken as 1 unit), no deterministic algorithm can achieve a ratio better than $O(\sqrt{n})$. Subsequently, Blum, Raghavan and Schieber [19] showed an algorithm for which the ratio matches this lower bound.

Thus, given no information about the scene, a deterministic algorithm cannot achieve a ratio better than $O(\sqrt{n})$. However, after making one trip that achieves this ratio the robot has some partial information about the scene. Can it exploit this information to improve its ratio on the second trip? Can it gain more information on the second trip and do even better on the third trip, and so on? It is important to note that partial information may not help improve competitiveness, if it is somehow not sufficiently relevant. In particular, merely using the one-trip procedure of Blum, Raghavan and Schieber on any trip may not necessarily reveal information that would help improve competitiveness on future trips. Thus the challenge is to stay optimally (up to constant factors) competitive on each trip given the information gained so far, and at the same time acquire information that will be useful in improving competitiveness on later trips. In other words, we would like each trip to be as short as possible given what the robot has found so far, and yet improve with the number of trips. (In particular we would not want a solution where the robot makes an artificially very long first trip and then subsequently "improves" on future trips.) This makes the multi-trip problem significantly harder than the single trip problem.

Results. We describe a deterministic strategy so that in any scene the length of the ith trip of the robot is guaranteed to be within an $O(\sqrt{n/i})$ factor of L, the length of the shortest s-t path. We also show that up to constant factors this is the best a deterministic strategy can do. These results appear in [17]. We have implemented X-Windows based code that simulates this and other navigation algorithms. A copy of the code may be requested by email from `chal@cs.cmu.edu`.

Our algorithm is based on a smooth *search-quality tradeoff:* we design an algorithm that given any $k \leq n$, searches a distance $O(L\sqrt{nk})$ and finds an s-t path of length at most $O(L\sqrt{n/k})$. An insight in this work is that a *tree* structure can be defined in the scene, where the nodes are portions of certain obstacles and the edges are "short" paths from a node to its children. The core of our algorithms is an on-line strategy for traversing this tree optimally.

Problem 2: Improving Performance in Paging

Chapter 4 presents a performance-improvement algorithm for the following paging scenario. Suppose a program repeatedly generates the *same* sequence of page requests. This might happen for instance when the program is inside a loop. A natural question to ask is, can one design a paging algorithm whose performance improves as the request sequence is repeated more times? This is non-trivial in many real computer systems, where the pager is *sleepy*, i.e. it only "wakes up" when there is a page fault, then decides which page to evict from fast memory, then goes back to sleep. Thus on each iteration the pager only knows the requests that resulted in page faults, so (much as in the navigation problem) it only has partial information about the request sequence. There is thus the possibility that the pager can accumulate information about the request sequence and improve its performance as the request sequence is repeated more times. The design of paging algorithms that improve their performance has been considered by others but in different contexts. [22, 41, 82, 64, 51].

Results. For the special case of this problem where k pages are in fast memory (or cache) and there are only $k+1$ pages total, we design an algorithm for a sleepy pager which has the following behavior when the same page request sequence σ is repeated. (We assume that on each iteration the pager knows the time relative to the start of the request sequence.) Let $OPT(\sigma)$ be the optimal (offline) number of page faults on σ. Then for each $t \leq k$, the average (i.e., amortized) number of page faults per iteration of the pager at the end of t iterations is within an $O(\frac{k}{t} \log t)$ factor of $OPT(\sigma)$, and for $t > k$, this average is within an $O(\log k)$ factor of $OPT(\sigma)$. Thus the average performance of the pager improves with the number of iterations t as long as $t \leq k$.

Problem 3: Online Learning of Switching Concepts

The thesis also considers in Chapter 5 a generalization of the traditional function-prediction problem from the on-line algorithms viewpoint. Most computational models of learning are suited to function-prediction tasks where the learner, after seeing samples of the input-output behavior of an unknown function (the "concept"), attempts to predict the value of the function at a new point.

In particular, Littlestone [58] introduced an *online learning model*, which can be viewed as the following game played between the learner and an adversary. The game is defined with respect to a concept class C defined on a domain X. Let us assume that C is a subclass of the class of 0-1-valued functions, such as DNF, or "monotone disjunctions". At the start of the game, the adversary picks a concept c from C and does not reveal c to the learner. In each round of the game, the learner chooses either to make a *prediction* or a *query*. In the former case, the adversary

gives the learner an example $x \in X$ and then the learner predicts 0 or 1, after which the adversary reveals the value of the label $c(x)$. In the latter case, the learner gives the adversary an example x, and the adversary responds with $c(x)$. One wants to design learning algorithms for which the total number of *mistakes plus queries* in any game (however long) is small. Such algorithms have been designed for various concept classes $\mathcal{C}$.

Suppose however that the labeling function is not fixed, but switches unpredictably between some small number k of unknown functions. In particular this switching may be history-dependent, in the sense that the function may stay the same for a while and then at some unpredictable time switch to another, and so on. For instance learner may be trying to predict the behavior of some system that can be in one of a small number of different states, and has a different behavior in each state. Or one can imagine trying to predict someone's movie preferences, where that person's preferences change depending on mood. We use the term *switching concepts* to refer to this kind of scenario. Ar, Lipton, Rubinfeld and Sudan [10] have considered a similar problem of learning from mixed data.

To study the online learning of switching concepts, we extend the above mistake bound with membership-queries (MBQ) model as follows. Initially, the adversary picks some set $S = \{c_1, c_2, \ldots, c_k\}$ of k concepts from $\mathcal{C}$, and keeps them secret from the learner. Each round of the game is the same as before, except that *before* the start of each round, the adversary picks some $c = c_i$ from S, and labels the example x with $c(x)$. We say the adversary has made a *switch* if the current c_i is different from the one used in the previous round.

It can be shown that even an algorithm that knows the k functions must make prediction mistakes proportional to the number of times the labeling function switches. Thus, just as in traditional online algorithms, a natural way to evaluate the quality of a prediction algorithm is to compare its cost with that of an algorithm that knows all k functions.

Results. For any two switching concepts c_1 and c_2 where $c_1 \neq c_2$ it is not hard to show that for any randomized algorithm (even one that knows both concepts), on any sequence of examples where the labeling concept switches s times, the *expected* number of mistakes is at least s. Our main result is the following. For the case that the two switching concepts are n-variable monotone disjunctions, we design an algorithm A such that for any $0 < \delta < 1$, the cost of A (i.e., total number of queries plus mistakes) is with probability at least $(1 - \delta)$ bounded by $p(n, \ln(1/\delta)) + s$, where p is a polynomial and s is the number of switches so far. We can thus say that the algorithm A is "1-competitive". The polynomial $p(n, \ln(1/\delta))$ is only an additive term, independent of s. This and other results appear in [16].

Summary

Learning is the improvement of performance at some task with experience. Most existing computational models of learning are only suited to tasks that can be naturally viewed as function approximation. This thesis extends the scope of learning theory; it introduces the design and analysis of algorithms for improving performance at other tasks, namely navigation and paging. Specifically, these algorithms are online algorithms which improve their competitiveness when repeatedly applied to the same problem. These algorithms must on each application not only be competitive, but also acquire useful information about the problem to improve their future competitiveness. This requirement leads to the design of novel kinds of online algorithms.

The thesis also applies competitive analysis to design an online algorithm for a natural extension of the standard function prediction problem, where the function labeling the examples switches unpredictably between two unknown functions.

Chapter 2

Performance-Improvement Algorithms

The next two chapters of this thesis are concerned with online algorithms whose performance (i.e., competitiveness) improves with repeated application to the same problem. In this chapter we present an abstract framework for studying such algorithms. We call our model a *hidden task system* since it is a variant of the task system model introduced by Borodin, Linial and Saks [23]. Our hope is that this general framework will provide insights as to how performance-improvement algorithms can be designed for other tasks besides those described in the next two chapters. We first briefly review the task system model of Borodin, Linial and Saks.

Task Systems

A task system consists of a pair (S, d) where $S = \{1, 2, \ldots\}$ is a (possibly infinite) set of states, and d is a state transition cost matrix. The matrix entry $d(i, j)$ is the (non-negative) cost of changing from state i to state j, when $i \neq j$. The task system (S, d) is called *metrical* if d is a metric (i.e., d is symmetric, satisfies the triangle inequality and $d(i, i) = 0$). For any task T, the cost of processing the task is a function of the state of the system; $T(j)$ denotes the cost of processing task T in state j. It is assumed that the system starts in a specified start state s_0. A *schedule* for processing a sequence of m tasks is a function σ where $\sigma(i)$ is the state in which the ith task is processed (and $\sigma(0) = s_0$). The cost of schedule σ on a task sequence $\mathbf{T} = T^1 T^2 \ldots T^m$ is the sum of all state transition costs and the task processing costs:

$$c(\mathbf{T}; \sigma) = \sum_{i=1}^{m} d(\sigma(i-1), \sigma(i)) \; + \; \sum_{i=1}^{m} T^i(\sigma(i)).$$

An online scheduling algorithm A for (S, d) receives a sequence of tasks $\mathbf{T} = T^1 T^2 \ldots T^m$

to be processed one at a time and the algorithm must output the state $\sigma(i)$ upon receiving T^i (that is, $\sigma(i)$ is a function only of the first i tasks in the sequence $\mathbf{T}$). The resulting schedule is denoted by $A(\mathbf{T})$. It is assumed that the online algorithm knows the state transition cost matrix d and that when the algorithm receives T^i, it *knows the cost of processing* T^i in each state of S. The cost of algorithm A on sequence $\mathbf{T}$, denoted $c_A(\mathbf{T})$ is defined to be $c(\mathbf{T}; A(\mathbf{T}))$. For a given task sequence $\mathbf{T}$, it is easy to compute (using dynamic programming) the cost $c_0(\mathbf{T})$ of an optimal schedule. An on-line algorithm A is said to be w-competitive [45] if there is a constant a such that

$$c_A(\mathbf{T}) \leq w c_0(\mathbf{T}) + a$$

for any finite task sequence $\mathbf{T}$.

In their paper, Borodin, Linial and Saks show a lower bound of $2s - 1$ on the competitive ratio of any deterministic online algorithm for an s-state metrical task system. They also show an algorithm that matches this bound. An important point to note is that the lower bound applies when arbitrary tasks are allowed, that is, for each task T the task processing costs $T(j)$ can be any non-negative number. Thus, although many particular online problems can be modeled as task systems, the above lower bound does not necessarily apply to those cases since the possible tasks are very special. One example is the paging problem with n pages and a k-page cache. The problem can be modeled as a task system where the states represent each of the $\binom{n}{k}$ cache configurations (i.e. the set of pages in the cache), and each task is a page request. The cost of "processing" page request i in a state is 0 if i is in the cache-set represented by the state, and it is ∞ otherwise. It is well known that a competitive ratio of k can be achieved for this problem, and this is much better than the lower bound of $\Omega(\binom{n}{k})$ when arbitrary tasks are allowed.

Hidden Task Systems

A *hidden task system* (S, d) is defined in exactly the same way as a metrical task system, except for the following new feature:

> The task processing cost functions $T()$ are only revealed in the states that the algorithm visits. In particular, when the algorithm receives task T to be processed, in order to find out the cost $T(j)$ of processing T in state j, the algorithm must move to state j. (Note that as in an ordinary task system the algorithm knows the state transition cost metric d.)

We show in Chapter 3 that a special case of the robot navigation problem can be viewed as a hidden task system. It will turn out that in this case the (integer) x-coordinates correspond to tasks and the (integer) y-coordinates correspond to states.

Since the robot is tactile, when it is at a point (x, y) it will not know whether or not there is an obstacle at a point with the same x-coordinate but a different y-coordinate y' unless it moves to the y coordinate y'. This makes the navigation model a hidden task system.

In Chapter 4 we study a paging model (which we call the sleepy model) where the pager is only aware of page faults. This is a hidden task system since unless the paging algorithm suffers a page fault, it does not know in which states (i.e. cache-configurations) the current task (i.e. page request) has cost ∞.

In our multi-trip robot navigation algorithms, it turns out that during an $s \rightarrow t$ trip the robot may often move "backward" toward s. In order to encompass such algorithms we must introduce the notion of a reversible task system: tasks can be "processed" both forward and backward, i.e., at any time the algorithm may choose to "undo" the last task it processed, in the following sense. If we define time i as the stage when the next task to be processed is task T^i, then when the algorithm at time i undoes the previous task T^{i-1}, it moves back to time $i-1$ (and so the "next task" at this stage would be T^{i-1}). Thus we require that all tasks that are undone must be "redone". The cost of undoing a task T in state j is $T(j)$, which is the same as the "forward" processing cost. (One could in general allow the forward and backward costs to be different, but we will not need to do this for our present purposes.) Thus at any time in a reversible task system the online algorithm may take one of three actions: (a) make a state transition, (b) process the next task, or (c) undo the previous task. In particular the algorithm may make several state transitions before processing or undoing a task.

A reversible task system is *hidden* if the online algorithm can find out the cost $T^i(j)$ of processing task T^i (the ith task in the sequence $T^1T^2\ldots$) only when it is in state j and the next task is either T^i or T^{i+1} (in the latter case we imagine the algorithm finds $T^i(j)$ when attempting to undo T^i).

We note in passing that one can also view the navigation model in terms of the *layered graph* traversal models studied by Papadimitriou and Yannakakis [70] and by Fiat *et. al.* [31]. A layered graph is an edge-weighted graph where there is a designated start vertex s and the other vertices are partitioned into sets ("layers") $L_1, L_2, L_3, \ldots$ and all edges run between adjacent layers in this sequence. An online layered graph traversal algorithm starts at s and moves along the edges of the graph. Its goal is to reach a vertex t in the last layer. In these layered graph models [70, 31], when the algorithm reaches a layer L_i, *all* edges to the next layer L_{i+1} are revealed. In the navigation problems, the layers correspond to integer x-coordinates, and the vertices within a layer correspond to integer y-coordinates. It is not hard to see that the navigation problems correspond to a variant of the layered graph model where (a) edges are allowed between vertices of the same layer (to allow motion along an x-coordinate) and (b) only the edges incident to the current vertex are revealed to

the algorithm (since the robot is tactile).

In a (possibly reversible) hidden task system (S, d), the cost $c_A(\mathbf{T})$ of an online scheduling algorithm A on a task sequence $\mathbf{T}$ is defined to be the sum of all the task processing/undoing costs and the state transition costs incurred by A in processing $\mathbf{T}$. As before, an online algorithm A is said to be w-competitive if there is a constant a such that

$$c_A(\mathbf{T}) \leq w c_0(\mathbf{T}) + a$$

for any finite task sequence $\mathbf{T}$, where $c_0(\mathbf{T})$ is the cost of processing $\mathbf{T}$ with an optimal offline scheduling algorithm that knows the task sequence $\mathbf{T}$ and the costs of processing each task in $\mathbf{T}$ in every state.

Note that when the online algorithm receives a task T it can visit each state j in S and find out $T(j)$. The algorithm would then have all the information available to an online algorithm in a standard task system, and it could output exactly the same schedule as such an algorithm. However, since there is a cost to moving between states, this may not in general be the most efficient way to process a given sequence of tasks. Thus in general the best competitive ratio achievable by an online algorithm for a hidden task system could be much worse than the corresponding quantity for a standard task system.

Improving with Repetition

After processing a given task sequence $\mathbf{T}_0 = T^1 T^2 \ldots T^m$ once, the online algorithm may know only some of the $T^i(j)$ values. Suppose that the *same* sequence $\mathbf{T}$ of tasks must be processed repeatedly. Since the scheduling algorithm A does not initially know any of the task processing costs, its competitive ratio might be quite large the first time. However, after processing the task sequence once, the algorithm has some *partial information* about the task processing costs. A natural question to ask is, can it exploit this partial information to improve its competitive ratio the second time? Can it gain even more information the second time, and do even better the third time, and so on? Is there a strategy so that the performance of the scheduler is (nearly) as good as possible given the partial information gained so far, and yet improves with the number of repetitions?

Let us make this more precise. For a fixed hidden task system (S, d), consider an online scheduling algorithm A that repeatedly processes the same sequence $\mathbf{T}$ of tasks some (unknown) number of times. Let $c_A^{(t)}(\mathbf{T})$ denote the total cost incurred by A after processing t repetitions of $\mathbf{T}$, and let $c_0(\mathbf{T})$ denote (as before) the optimal offline cost of processing $\mathbf{T}$ once. Let $s = |S|$ and let $|\mathbf{T}|$ denote the number of tasks in $\mathbf{T}$. Then the *average* (i.e., amortized) *competitive ratio* of A at the end of t repetitions

is defined as

$$\rho(A,m,s,t) = \sup_{\substack{|\mathbf{T}|=m \\ c_0(\mathbf{T})\neq 0}} \frac{c_A^{(t)}(\mathbf{T})}{t \cdot c_0(\mathbf{T})}.$$

One way to formally phrase the questions in the previous paragraph is to ask whether there is an algorithm A such that for every t the ratio $\rho(A,m,s,t)$ is as small as possible. Can a general lower bound on $\rho(A,m,s,t)$ be shown? Note that merely requiring the average competitive ratio to be small might allow the algorithm to process some repetitions unusually expensively while compensating for this cost on other repetitions. A somewhat more stringent requirement is to insist that for each i, the competitive ratio on the ith repetition be small. If we let $c_A(i,\mathbf{T})$ denote the cost of algorithm A on the ith repetition, we can define the competitive ratio on the ith repetition as

$$\rho_i(A,m,s) = \sup_{\substack{|\mathbf{T}|=m \\ c_0(\mathbf{T})\neq 0}} \frac{c_A(i,\mathbf{T})}{c_0(\mathbf{T})}.$$

We can then ask whether there is a strategy A such that for each i the ratio $\rho_i(A,m,s)$ is small.

It would be interesting to answer some of the above questions for general hidden task systems. As a first step, in this thesis we answer these questions for two particular hidden task systems. As we mentioned before, the navigation and paging problems correspond to hidden task systems with a special state transition metric d and a special set of possible tasks (i.e. task processing costs). We will show lower bounds on the ratio $\rho(A,m,s,t)$ for these hidden task systems. It turns out that these lower bounds decrease with increasing t. We develop algorithms A for which $\rho(A,m,s,t)$ for each t is close to this lower bound. Our algorithms also have the property that their average competitive ratio $\rho(A,m,s,t)$ decreases with increasing t. In the case of the navigation problem we also show an algorithm A for which $\rho_i(A,m,s)$ is an optimally (up to constant factors) decreasing function of i.

In each of the next two chapters we first describe a particular performance-improvement algorithm, and then restate our result in the language of hidden task systems.

Chapter 3

Improving Performance in Navigation

3.1 Introduction

Imagine you have just moved to a new city; you are at your home and you want to go to your office, but you do not have a map. Let's assume you know your coordinates and those of your office. Since you do not have a map, your first trip to the office could turn out to be much longer than the shortest path to get there. However on your way to the office you saw part of the city, so perhaps you can use this information to do better when you head back home. You would like to perform as well as possible on each trip and yet improve your performance with experience. You do not want to spend too much time exploring the city on any one trip. In other words, you want a strategy so that on each trip, (a) your path is (nearly) the shortest path you can find based on the information you have so far, and (b) you gain useful information for improving later trips.

Such a problem can arise naturally in robotics: A robot exploring Mars might repeatedly transport samples from one point to another on the planet, and it must circumvent unknown obstacles when making these trips. One would like the robot to make each trip as quickly as possible, and also improve its path as it makes more trips.

In this chapter we consider a scenario (examined in [19, 46, 70]) where the start point s and target t are in a 2-dimensional plane filled with non-overlapping axis-parallel rectangular obstacles. A tactile, point robot begins at s, and knows its current position and that of the target, but it does not know the positions and extents of the obstacles; it only finds out of their existence when it touches them. In the problem considered in previous papers, the robot must make a single trip from s to

t circumventing the obstacles. We call this the *one-trip* problem. In this chapter we consider a robot that may be asked to make *multiple trips*, going back and forth between s and t.

It is intuitive that in the multiple trips problem, information from previous trips must be used if one hopes to improve the path later on. In particular, on later trips some but not all of the obstacles in the scene are known. We therefore have a type of on-line problem that is different from the standard scenario, in that *partial information* about the future (e.g., about obstacles that lie ahead) must be exploited to achieve good performance.

A particular arrangement of s, t, and the obstacles is called the *scene*. Let n denote the *Euclidean* distance between s and t in the scene, where the obstacles have width and height at least 1. Papadimitriou and Yannakakis [70] show for the one-trip problem a lower bound of $\Omega(\sqrt{n})$ on the ratio of the distance traveled by the robot to the actual shortest path (the *competitive ratio*) for any deterministic algorithm. Blum, Raghavan, and Schieber [19] describe an algorithm whose performance matches this bound. Whether randomization can help improve upon this bound is an open question, although a lower bound was obtained by [46].

For the multiple trips problem, there are several ways one might formalize the intuitive goal described in the first paragraph. One natural way is to consider the total distance traveled in the k trips between s and t and examine the ratio of this to k times the shortest path. Thus, for $k = 1$, the previous results give a ratio of $\Theta(\sqrt{n})$. For $k = n$ it is not hard to see that one can achieve a ratio of $O(1)$ by simply performing a search (of cost proportional to n times the shortest path length) to find the shortest path on the first trip. Our main result is to show an optimal smooth transition. For $k \leq n$ we present a deterministic algorithm whose competitive ratio is $O(\sqrt{n/k})$ and give an $\Omega(\sqrt{n/k})$ lower bound for deterministic algorithms. A key idea of the algorithm is to optimally traverse a certain *tree* structure based on the obstacles in the scene.

Notice that the "cumulative" formulation allows one to search "hard" on the first trip to find a short path, and then use this short path on the remaining $k-1$ trips, which is what we do. This result can be thought of as one that shows how to optimally trade off exploration effort with the goodness of the path found. We in addition show how to modify the algorithm so that on the ith trip, its ratio *for that trip* is $O(\sqrt{n/i})$. So, this algorithm is optimal for each prefix cumulative cost and in addition does not spend too much effort on any one trip. Thus, the algorithm can be viewed as one that optimally improves its performance with each trip, achieving the intuitive goal described at the start of this chapter.

Since there is a large body of research related to the work described here, we postpone its discussion to Section 3.11 for the sake of continuity.

3.2 The model, and some preliminaries

We consider two dimensional scenes containing non-overlapping rectangular obstacles such that two given sides of any two rectangles are either parallel or perpendicular to each other. A tactile, point robot is required to travel between two points s and t in the scene. In such a scene we define an x-y rectangular coordinate system where s is the origin $(0,0)$ and the x-axis is parallel to the side of some rectangle. (Thus the rectangular obstacles are "axis-parallel"). Let $\mathcal{S}(n)$ denote the class of scenes with axis-parallel rectangular obstacles where the Euclidean distance between s and t is n. As mentioned above, we assume that the width and height of the obstacles are at least 1 (this in essence defines the units of n) and for simplicity that the x-coordinates of the corners of obstacles are integral. Thus no more than n obstacles can be placed side by side between s and the x-coordinate of t. We assume that when obstacles touch, the point robot can "squeeze" between them.

To simplify the exposition, we take t to be the infinite vertical line (a "wall") $x = n$ and require the robot only to get to any point on this line; this is the Wall Problem of [19]. Our algorithms can be easily extended to the case where t is a point, using the "Room Problem" algorithms of [19] or [13]. This modification is described in Section 3.8.

As mentioned above, we assume that the only sensing available to the robot is tactile: that is, it discovers an obstacle only when it "bumps" into it. It will be convenient to assume that when the robot hits a rectangle, it is told which corner of the rectangle is nearest to it, and how far that corner is from its current position. As in [19], our algorithms can be modified to work without this assumption with only a constant factor increase in cost (i.e., distance traveled). In particular, the results in [12] imply that if the nearest corner is distance d away then the robot can determine this by walking a distance at most $2 + 9d$.

Consider a robot strategy R for making k trips between s and t. Let $R_i(S)$ be the distance traveled by the robot in the ith trip, in scene S. Let $L(S)$ be the length of the shortest obstacle-free path in scene S between s and t. We define the *cumulative* k-trip competitive ratio as

$$\rho(R, n, k) = \sup_{S \in \mathcal{S}(n)} \frac{R^{(k)}(S)}{kL(S)},$$

where $R^{(k)}(S) = \sum_{i=1}^{k} R_i(S)$ is the total distance traveled by the robot in k trips. That is, $\rho(R, n, k)$ is the ratio between the robot's total distance traveled in k trips, and the best k-trip distance. We define the *per-trip* competitive ratio for the ith trip as

$$\rho_i(R, n) = \sup_{S \in \mathcal{S}(n)} \frac{R_i(S)}{L(S)}.$$

Our main results are the following. First, we show for any k, n, and deterministic algorithm R, that $\rho(R,n,k) = \Omega(\sqrt{n/k})$. Second, we describe a deterministic algorithm that given $k \leq n$ achieves $\rho(R,n,k) = O(\sqrt{n/k})$. Finally, we show an improvement to that algorithm that in any scene achieves $\rho_i(R,n) = O(\sqrt{n/i})$ for *all* $i \leq n$.

The remainder of the chapter is organized as follows. The next section presents the lower bound. Sections 3.4, 3.5 and 3.6 describe the algorithm and its modifications. Section 3.10 discusses open problems, and we conclude with related work in Section 3.11.

3.3 A Lower Bound for k Trips

In this section we show that for any deterministic robot strategy R for making k trips between a point s and a line t (distance n from s), a scene can be constructed using axis-parallel rectangular obstacles such that for some constant c, its average trip length is at least $cL\sqrt{n/k}$, where L is the length of the shortest s-t path. In particular, this lower bound applies regardless of how the robot moves on each trip (For instance the robot may move "backward" towards s on an $s \rightarrow t$ trip).

Theorem 1 (k-trip Cumulative Lower Bound) *For $k \leq n$, the ratio $\rho(R,n,k)$ is at least $\Omega(\sqrt{n/k})$, for any deterministic algorithm R.*

Proof: Since R is deterministic, an adversary can simulate it and place obstacles in S as follows. Recall that s is the point $(0,0)$.

Consider a set of obstacles of fixed height $h \geq \sqrt{n}$ and width 1, in a full "brick pattern" on the entire plane, as shown in Fig. 3.1, with s at the center of the left-side of an obstacle. (Recall that the point robot can "squeeze" between bricks). The adversary simulates R on this scene, notes which obstacles it has touched at the end of k trips, then removes all other obstacles from the scene. This is the scene that the adversary creates for the algorithm, and say it contains M obstacles. Since the robot does not know about obstacles that it did not hit, it will walk exactly the same way on this M-obstacle scene as it would on the full brick pattern. The brick pattern ensures that R must have hit at least one brick at every integer x-coordinate, so $M \geq n$. Further, this arrangement forces the robot to hit a brick (and pay vertically $h/2$) at every integer x-coordinate on *every* trip. Therefore its total k-trip distance $R^{(k)}$ is at least $nkh/2$.

We first argue that $R^{(k)}$ is at least *linear* in M. Imagine the full brick pattern to be built out of four kinds of bricks (red, blue, yellow and green, say) arranged in a periodic pattern as shown in the figure. This arrangement has the following property:

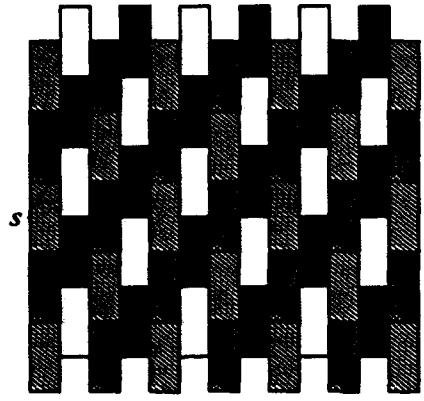

Figure 3.1: A full brick pattern with bricks of 4 colors.

for each color, to go from a point on one obstacle of that color to a point on any other of the same color the robot must move a distance at least $h/2$. Out of the M obstacles hit by the robot, at least $M/4$ must have the same color, say blue. So regardless of how the robot moved, since it has visited $M/4$ blue obstacles, we have $R^{(k)} \geq Mh/8$, which implies $M \leq 8R^{(k)}/h$.

Using an argument similar to that used in the one-trip lower bound proof of [70], we now show that there is a path from s to the wall whose length is at most proportional to the *square-root* of M. We claim there is a non-negative integer $j \leq \sqrt{M}$ such that at most $\sqrt{M}$ obstacles have centers at the y-coordinate jh. This is because a given obstacle intersects at most one y-coordinate of the form jh, and there are M obstacles. Thus, there is a path to t that goes vertically to the y-coordinate jh, then horizontally along this y-coordinate, going around at most $\sqrt{M}$ obstacles. The total length of this path is at most $h\sqrt{M}+h\sqrt{M}+n$, which is at most $3h\sqrt{M}$ since $n \leq M$ and $\sqrt{n} \leq h$. Since $M \leq 8R^{(k)}/h$, this path is in fact of length at most $3\sqrt{8hR^{(k)}}$. Thus the k-trip ratio is at least $R^{(k)}/(3k\sqrt{8hR^{(k)}})$. Recalling that $R^{(k)} \geq nkh/2$, this is at least $\frac{1}{12}\sqrt{n/k} = \Omega(\sqrt{n/k})$. ■

This lower bound implies that no deterministic algorithm R can guarantee that the per-trip competitive ratio $\rho_i(R,n)$ is $o(\sqrt{n/i})$ for every i. Indeed, if such an algorithm existed then its cumulative competitive ratio at the end of k trips would be

$$\frac{1}{k}\sum_{i=1}^{i=k} o(\sqrt{n/i}) \;=\; \frac{1}{k}o(\sqrt{nk}) \;=\; o(\sqrt{n/k}),$$

which violates the above lower bound. In this sense a per-trip ratio of $O(\sqrt{n/i})$ is the best a deterministic algorithm can achieve.

It is not hard to see that this lower bound also holds for the case where t is a point rather than a wall.

3.4 The Optimal k-trip Cumulative Algorithm

Let us assume for simplicity that the algorithm knows the length L of the shortest obstacle-free path from s to t. In Subsection 3.5.3 we show how this assumption can be removed by using a standard "guessing and doubling" trick. Note that the shortest obstacle-free s-t path must lie entirely within a *window* of height $2L$ centered at s, since any s-t path that leaves the window must be longer than L. In the remainder of this chapter, let us also assume that we are designing a left-to-right (i.e. s to t) trip of the robot; we can just assume that upon reaching t the robot retraces the path it just followed in getting to t from s.

This last observation immediately leads to an easy algorithm to achieve a cumulative k-trip ratio of $O(1)$ for $k = n$:

> **First trip:** Using say a depth-first-search, explore the entire rectangular window of height $2L$ and width n (centered at s) between s and t. This can be done by walking a total distance of $O(Ln)$. Compute the shortest obstacle-free s-t path (of length L).
>
> **Remaining $n-1$ trips:** Use the shortest path.

Clearly the average trip length is $O(L)$, so that the cumulative n-trip ratio is $O(1)$.

Thus the cases $k = 1$ (the Blum-Raghavan-Schieber, or BRS algorithm) and $k = n$ are easy. The main difficulty is in dealing with intermediate values of k. Actually, our high-level optimal cumulative strategy for k trips is similar to the n-trip algorithm just described:

> **First trip:** Perform an "exploratory" walk of length $O(L\sqrt{nk})$, in such a way that an s–t path P of length $O(L\sqrt{n/k})$ can be composed out of portions of the walk;
> **Remaining $k-1$ trips:** Use the path P.

Clearly the average trip length is $O(\frac{1}{k}(L\sqrt{nk} + (k-1)L\sqrt{n/k})) = O(L\sqrt{n/k})$, so the cumulative k-trip ratio is $O(\sqrt{n/k})$. In other words, in the first trip, the robot travels a distance of $O(\sqrt{k})$ times the $L\sqrt{n}$ bound for the one-trip problem, and finds an s-t

path that is guaranteed to be $\Omega(\sqrt{k})$ times shorter than $L\sqrt{n}$. The main challenge is to design the "search" algorithm for this trip; this algorithm must smoothly tradeoff search effort for the quality of the path found.

In order to make clear the main ideas, in the next section we describe this algorithm for a class of scenes we call *simple scenes.* We first describe the one-trip BRS algorithm, discuss why a naive extension of their algorithm does not work for k trips, and finally present the search algorithm. In Section 3.6 we show how to extend this algorithm to handle scenes with arbitrary axis-parallel rectangular obstacles. In Section 3.7 we describe how to modify this algorithm to a more incremental algorithm whose ith trip is guaranteed to be within a $O(\sqrt{n/i})$ factor of the optimal path, for each $i \leq n$. Section 3.8 describes the modification of our algorithm for point-to-point navigation.

In the remainder of this chapter we will use the words up, down, left, and right to mean the directions $+y, -y, -x, +x$ respectively. When we say point A is above, below, behind, or ahead of a point B we will mean that A is in the $+y, -y, -x, +x$ direction respectively from B. Finally, vertical (horizontal) motion is parallel to the y (respectively, x) axis. At any stage, the current coordinates of the robot (which are known to the robot) are denoted by (x, y).

3.5 The Algorithm for Simple Scenes

It turns out that certain scenes that we call *simple scenes* capture most of the difficulties in designing online navigation algorithms (for both the one-trip and k-trip problems). A scene is called simple if (a) all obstacles have the same height $2h$ and width 1, and (b) the lower left hand corners of all obstacles are at coordinates of the form (i, jh) for integer i and j. Thus in a simple scene the bottom sides of any two obstacles are either at the same y-coordinate, or at y-coordinates that differ by h. For instance, the obstacles in Fig. 3.2 form a simple scene. Observe that in a simple scene is that one can move vertically along any integer x-coordinate without encountering any obstacles.

Very roughly speaking, scenes where obstacle heights and positions are arbitrary can be treated as simple scenes where the obstacles are fixed-height *portions* of the actual obstacles. As we point out later, dealing with arbitrary obstacle *widths* causes some messiness in our algorithm.

3.5.1 Fences and the One-trip BRS Algorithm

For both the one-trip BRS algorithm and the k-trip algorithms, the notion of a *fence*[1] is important. An *up fence* is a sequence of obstacles extending across a window of height $2L$ that is centered vertically at s, where each obstacle in the sequence is exactly h higher and to the right of the preceding obstacle in the sequence. (see Fig.[2] 3.2). By "extending across" the window we mean that the lowest obstacle in the fence is centered at or below the bottom of the window, and highest obstacle is at or above the top of the window. Henceforth, whenever we say "the window" we will mean the window of height $2L$ centered vertically at s. [3] Between the top half of the left side of an obstacle and the bottom half of the left side of the next obstacle in the fence, there is a rectangular region of height h, which we call a *band*. A fence can thus be viewed as a contiguous sequence of bands extending across the window. (See Fig. 3.2).

A point P is said to be *left* of a fence F if an imaginary horizontal line from P to the right intersects the *top half* of some obstacle of F. On the other hand, P is said to be *right* of F if this horizontal line does not intersect any obstacle of F. A path is said to *cross the fence* if it connects some point left of the fence to some point that is right of the fence, *and* stays inside the window. (The definitions for a down-fence are analogous). For instance the path from A to B in Fig. 3.2 crosses the fence shown. Any path that crosses a fence therefore pays vertically at least h to cross a fence. If there is a fence in the scene whose leftmost obstacle is to the right of s, then clearly any s-t path that stays inside the window must cross this fence, and pay vertically h to do so (see Fig. 3.2).

It is easy to find a fence with vertical cost at most $2L$. For instance, starting from the bottom of the window, an up fence can be found as follows:

> **Repeat until** at top of the window (i.e. $y = +L$): walk to the right till an obstacle is hit, then move up to the top of the obstacle.

The one-trip BRS algorithm restricted to simple scenes is then essentially this:

> Initially, walk from s down to the bottom of the window. Until the wall is reached, walk to the right, alternately building up and down fences across a window of height $2L$ centered at s.

[1] What we call a fence here is similar to a *sweep* in [19]

[2] Since $L \geq n$, the window in this figure should really be taller than its width. However for the sake of clarity, in this and all figures in this chapter the vertical dimension has been compressed considerably.

[3] We later find it convenient to allow fences to extend significantly beyond the window boundaries; the only requirement is that the fence extend *at least* across the window.

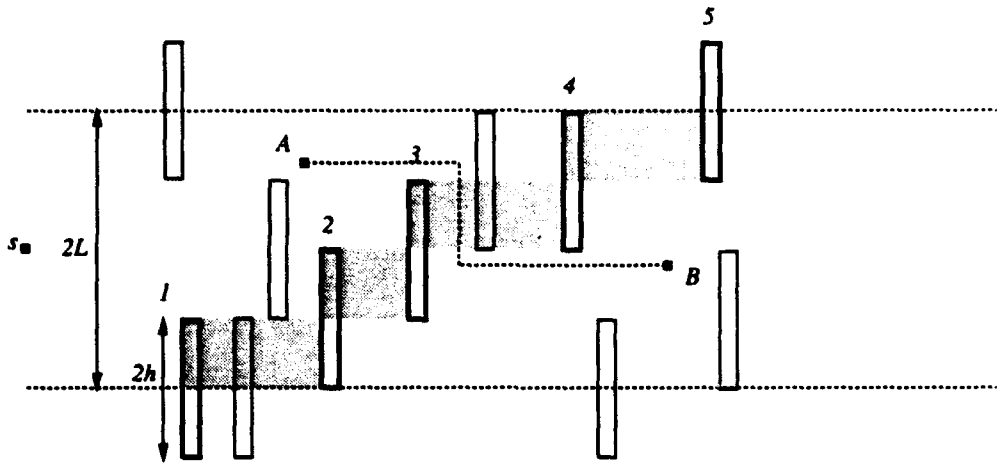

Figure 3.2: A fence in a simple scene. The obstacles $\langle 1,2,3,4,5\rangle$ with thick boundaries form a fence. The dashed line connecting points A and B crosses the fence. The shaded regions are the bands of the fence.

The robot never walks backward, so its total horizontal cost is n, and since $L \geq n$, this cost is only a small order term. Note that every obstacle hit by the robot is part of *some fence*. Thus every time the robot spends $2L$ (vertically) to build a fence, it is also forcing the optimal offline path to spend at least h to cross the fence. So if $h \geq L/\sqrt{n}$ clearly the competitive ratio is $O(\sqrt{n})$. The case $h < L/\sqrt{n}$ is even easier to handle: the robot hits at most n obstacles (since they have width 1 and the robot never walks backwards), so its total vertical cost is at most $nh < L\sqrt{n}$.

3.5.2 A Naive k-trip Approach

For the k-trip problem an intuitively reasonable approach is to extend the BRS idea as follows:

> On each trip, make new fences that are *disjoint* from previous fences.

We say that two fences are disjoint if the bands of one fence do not intersect the bands of the other; therefore if the obstacles of the fences are at x-coordinates between those of s and t, any s-t path that stays inside the window must pay (vertically) at least h to cross *each* fence. The reason for considering only disjoint fences is that if the bands overlap, then a path can cross both fences with a vertical cost of only h by crossing the region common to the bands. Clearly two disjoint fences cannot share obstacles. Here is a simple way to test whether two up-fences F and F' are disjoint, where the first (lowest) obstacle of F' is to the right of the first obstacle of F. Imagine

that the obstacles of F and F' are the only obstacles in the scene. Then F and F' are disjoint if and only if each obstacle of F' can be translated to the right without hitting [4] an obstacle of F. Thus in Fig. 3.3, $F_1 = \langle 1,2,3,4,5\rangle$, $F_2 = \langle 6,7,8,9,10\rangle$, and $F_3 = \langle 6,7,P',9,10\rangle$, are all legal fences, but F_1, F_2 are disjoint, while F_1, F_3 are not (obstacle P' hits 4 when it is translated to the right, and the band of F_3 between P' and 9 overlaps the band of F_1 between 3 and 4). Indeed, since F_1, F_3 are not disjoint it is possible to "sneak" past both fences through the gap between obstacles P' and 4, paying only h.

Suppose that one could find new disjoint fences "cheaply" (roughly $2L$ cost), *and* one could cross old fences cheaply (roughly h cost). Then the average online cost *per trip* to get past a fence is $O(L/k)$ ($O(L)$ to build it once, and h to cross it up to $k-1$ times), whereas the offline optimal path pays at least h to cross it. Let $\tau = L/\sqrt{nk}$. If $h \geq \tau$ we would achieve a cumulative ratio of $O(\frac{L/k}{L/\sqrt{nk}}) = O(\sqrt{n/k})$. As before the $h < \tau$ case is easy: the total vertical cost on each trip is at most $nh < L\sqrt{n/k}$. In fact if $h < \tau$ it is easy to see that the simple strategy that just walks to the right along some y-coordinate, (going around any obstacles on the way) achieves an $O(\sqrt{n/k})$ ratio on *each* trip.

However, on the second and later trips, if the robot attempts to find fences in the straightforward manner described above, it could cost a lot more than $O(L)$ to keep the fences disjoint from previously discovered fences. Consider the scene in Fig. 3.3. Suppose that the top fence $\langle 1,2,3,4,5\rangle$ was created on the first trip, and the robot is trying to create a new fence on the second trip, starting at obstacle 6, using the simple procedure described before. It is clear that this will result in the fence $\langle 6,7,P',4,5\rangle$ which "collides" and merges with the first fence. Note that P' is the first "invalid" obstacle encountered when building the second fence: when translated to the right, it hits obstacle 4 of the first fence. To ensure disjointness, the third obstacle of the second fence must be to the right of the obstacle 4 in the first fence.

The robot could recover from the collision by trying to find an obstacle P_1' to the right of obstacle 4 that can be used in place of P'. One way to do this is to go up to obstacle 4 (perhaps by joining the path previously used to get to obstacle 4), then down along 4, then to the right (see Fig. 3.4). The next obstacle encountered can then be included in the second fence.

However in general such recovery strategies could be expensive. Consider the scene shown in Fig. 3.5. After the fence $\langle 1,2,3,4\rangle$ is built on the first trip, in the next trip the robot starts building a new fence [5] at obstacle 5, and encounters an invalid obstacle 6. To find the replacement (7) the robot must move vertically along *two* obstacles: 6 and 2. Similarly on the third trip, the invalid obstacle 13 is replaced by

[4] The top of an obstacle "grazing" the bottom of another obstacle is not considered a "hit"

[5] For convenience, we show each fence starting h lower than the preceding fence.

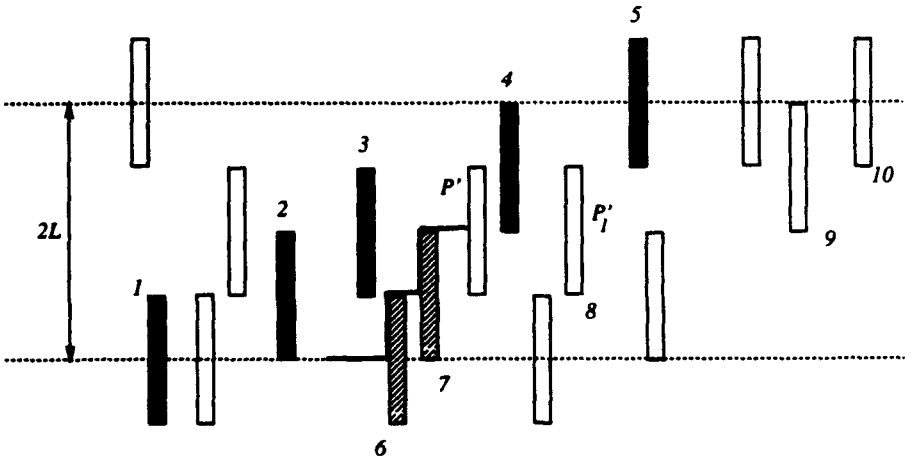

Figure 3.3: The robot encounters an invalid obstacle P' when finding the second fence: no continuation of the partial fence $\langle 6, 7, P' \rangle$ can be disjoint from the first fence $\langle 1, 2, 3, 4, 5 \rangle$.

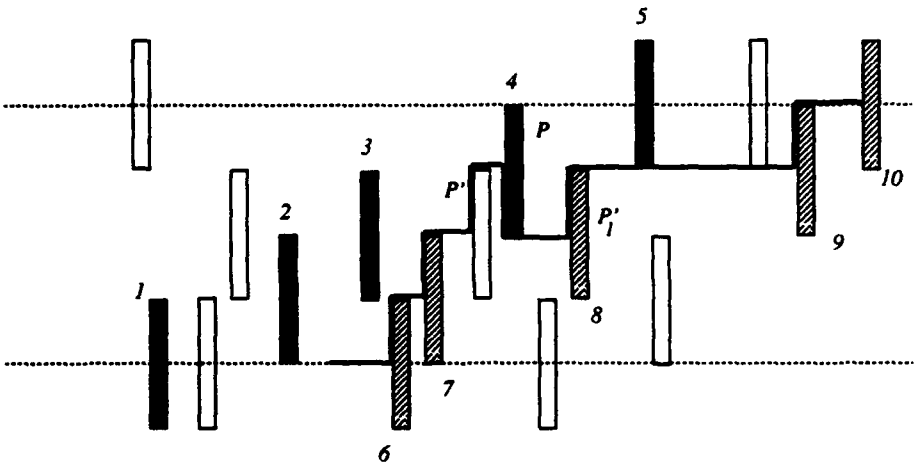

Figure 3.4: A possible way to recover from the collision of the previous figure: find an obstacle P'_1 (obstacle 8) to use in place of the invalid obstacle P', and continue building the second fence.

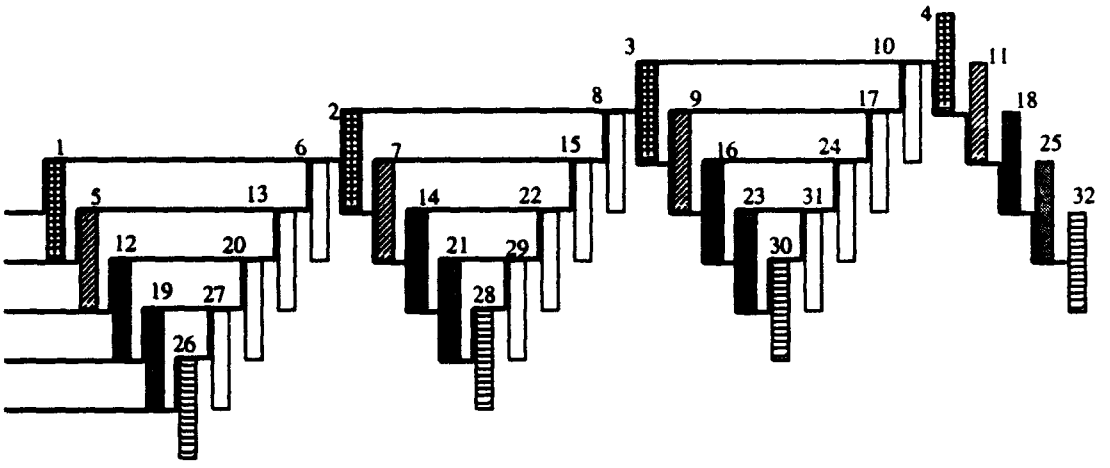

Figure 3.5: A scene where the naive repair strategy is too expensive. Five fences are created in succession. Obstacles are numbered in the order in which they are visited. Similarly-shaded obstacles constitute a fence. Unshaded obstacles are not part of any fence.

14, after moving along *four* obstacles 13,6,2,7. (There could be other obstacles – not shown in the picture – in the space between 13 and 14, so that *all paths* from 13 to 14 involve moving along 4 obstacles.) Thus on the ith trip a single repair involves vertical motion along $2(i-1)$ obstacles. It is not hard to see then that building u fences this way will cost $\Omega(u^2L)$ and not $O(uL)$.

It is possible that other more sophisticated ways of recovering from the fence collision problem are not this expensive, but we do not know of any.

3.5.3 Groups of Disjoint Fences

Our approach is to give up entirely on trying to create disjoint new fences on each trip. Specifically, instead of trying to build single fences on *each* trip, we do the following. Suppose that $h = aL/\sqrt{nk}$ for some $a \geq 1$ (recall that $2h$ is the obstacle height, and $a < 1$ is an easy case to handle). Then our strategy is:

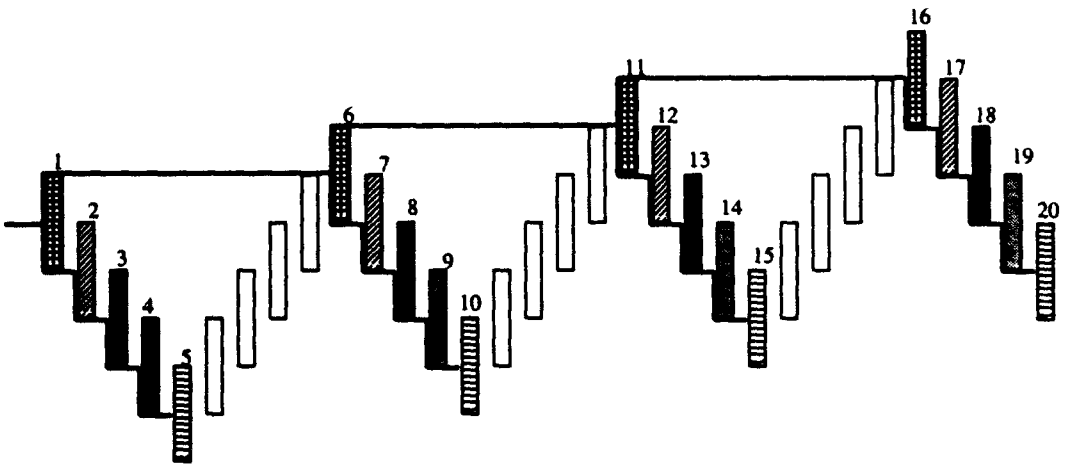

Figure 3.6: The same scene as in the previous Figure. Five fences are created in tandem. Obstacles are numbered in the order in which they are visited. Note the considerably smaller vertical distance walked.

First trip: Build *groups* of $G = \lceil \frac{k}{a} \rceil$ disjoint fences (alternately upward and downward) across a window of height $2L$ centered at s, until the wall is reached. Ensure that

(a) the cost of building each group is $O(kL)$,

(b) an $O(L)$ length path crossing each group (i.e. going from the x-coordinate of the leftmost obstacle of the leftmost fence to the x-coordinate of the rightmost obstacle of the rightmost fence) is found, and

(c) the right end of each group-crossing path is the left end of the next group-crossing path.

Remaining $k-1$ trips: Follow the group-crossing paths to the wall.

The first trip is shown schematically in Fig. 3.7. To see why the above strategy achieves a $O(\sqrt{n/k})$ ratio, assuming we can somehow satisfy (a), (b) and (c), notice that the *average* online cost per trip to get past each fence group is $O(L)$ (average of $O(kL)$ building cost on the first trip, and $O(L)$ crossing cost on the remaining trips). Crossing each group costs the optimal offline path at least $(k/a)h = L/\sqrt{n/k}$, so the average online trip length is within a $O(\sqrt{n/k})$ factor of optimal.

A Search-Quality Tradeoff. It is easy to see that the first trip achieves the search-quality tradeoff mentioned earlier. Since each fence group costs the offline optimum

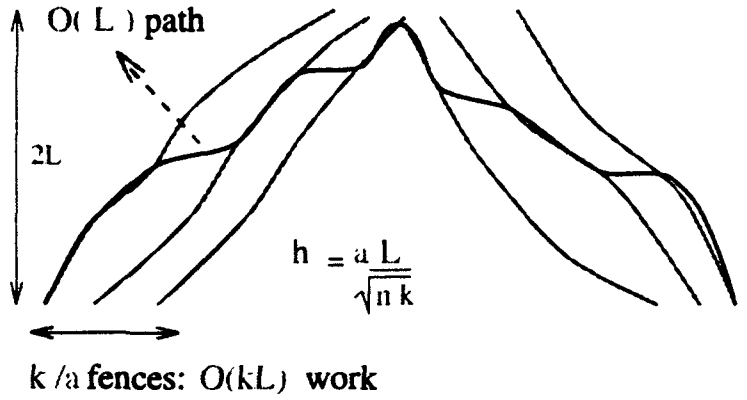

Figure 3.7: High-level view of the optimal k-trip strategy. First trip: create groups of fences with short group-crossing paths. Remaining trips: follow these short paths.

path at least $L/\sqrt{n/k}$ to cross, the robot cannot find more than $\sqrt{n/k}$ groups before reaching the wall. Thus the total length of its first trip is $O(kL\sqrt{n/k}) = O(L\sqrt{nk})$, and the total length of the group-crossing paths is $O(L\sqrt{n/k})$.

The Doubling Strategy when L is unknown. Note that if L is not known, we can just guess a value, and if the above steps repeat more than $\sqrt{n/k}$ times we can double our guess and repeat the entire procedure. Thus there is only a constant factor penalty for not knowing L.

The advantage of building the fences in a group is that it is not necessary to build a complete fence before finding another one (and we just saw that this could be very expensive). Rather, the robot can judiciously skip from working on a partial fence to work on another partial fence, thus extending the frontiers of the fences roughly concurrently. For instance, the 5 fences in Fig. 3.5 can be constructed as a group in the manner shown in Fig. 3.6. The robot first finds the lowest obstacle of each fence, then the the second obstacle of each fence, and so on. (Because of the regularity of the scene, it is efficient to build the fences exactly at the same rate; in general our algorithm may add several obstacles to a fence before skipping to another fence) It is worth observing here that although the ith obstacle of a fence may not be "easily" reachable from the $(i-1)$st obstacle of the same fence (which is why it is expensive to build the complete fences in succession), it *is* easily reachable from the ith obstacle of the fence *above* (by a simple path that goes down h and then to the right). We will show later that we can define fences so that we can always guarantee that the

ith obstacle of a fence is easily reachable either from the preceding obstacle on the same fence or the corresponding obstacle of the fence above.

Examples of Search Trips

The search algorithm will be described later but to give a feel for it we show examples of search trips for $n = 70$ and $k = 1, 2, 3$ and 6 in Figures 3.8, 3.9, 3.10, 3.11. These are screen dumps of an X-windows based program for simulating various navigation algorithms on simple scenes. [6] In the simulation, the user can generate a simple scene using mouse clicks, and select one of four algorithms: (a) RAND: a randomized strategy; whenever the robot hits an obstacle, it goes up with probability 0.5 and down with probability 0.5, (b) BRUT: this is the brute force strategy; whenever the robot hits an obstacle, it goes around the obstacle and then to the other side to the same y-coordinate as the start point, (c) BRS: the Blum-Raghavan-Schieber single-trip algorithm, and (d) TREE: the search trip for a k value that can be chosen by the user. (The search algorithm is called TREE for reasons that will become clear later) The four figures show only the TREE algorithm. The point s is at the left center of the scene, and t is the right side of the scene. The obstacles touched by the robot are highlighted. In each group of fences, even and odd-numbered fences are shaded differently so the fences are easier to distinguish. The "Work" window shows the total distance walked (the thin line) in the form $\alpha x + \beta y$ where α is the horizontal distance in units of obstacle-widths and β is the vertical distance in units of half-obstacle heights (i.e. h). The "Path" window shows the length of the s-t path found (the thick line) in the same format. As before, in these examples the vertical dimension has been compressed for clarity; each obstacle must be really thought of as being (say) 10 times taller than its width.

In the figures as k ranges through 1,2,3,6, the robot works "harder" and finds paths whose vertical lengths are 38, 29, 23, 24 respectively. Thus in a particular scene, the length of the path found need not always improve as k increases. The only guarantee is that the *worst case* path length decreases with increasing k, i.e., for some constant c, on any scene where the s-t distance is n, the length of the path found will not exceed $cL\sqrt{n/k}$ (and the total work will not exceed $cL\sqrt{nk}$).

3.5.4 Fence-trees

We would like to build a collection of $G = \lceil \frac{k}{a} \rceil$ up-fences (say) across a window of height $2L$, with a cost of $O(kL)$ and find an $O(L)$ length path crossing the collection. Our key idea is to define a *tree* structure whose nodes are certain obstacles in the scene,

[6] A copy of the code for this demo can be obtained by sending email to `chal@cs.cmu.edu`

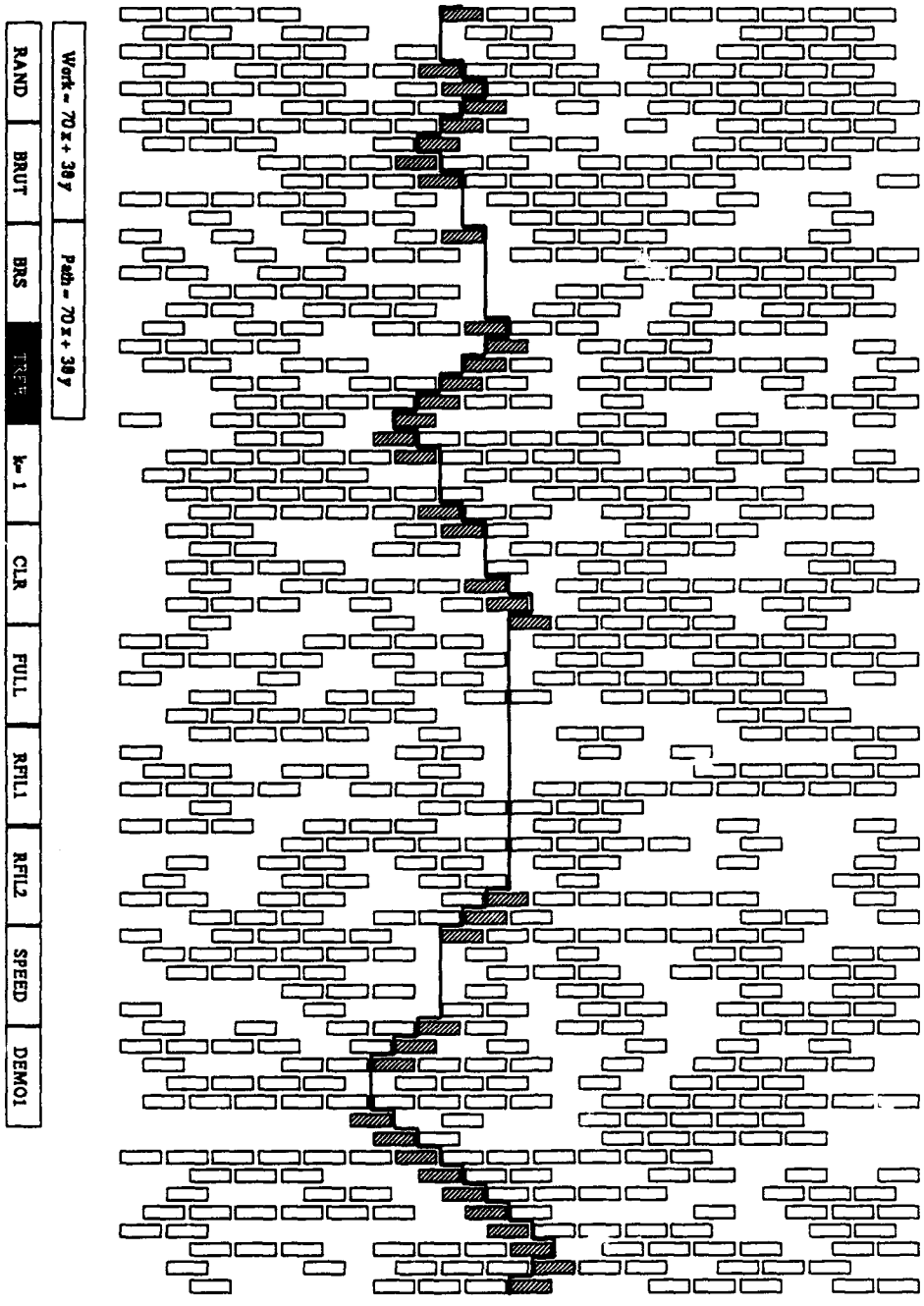

Figure 3.8: Initial search trip, $k = 1$. This and the next three figures show the initial search trip of the robot, for $k = 1, 2, 3$ and 6 respectively. The thin line shows the path of the robot, the thick line shows the s-t path found (s is at center left, t is at the right). In each figure, groups of k fences (alternately up and down) are found with $k, 2k, 3k, \ldots$ obstacles each.

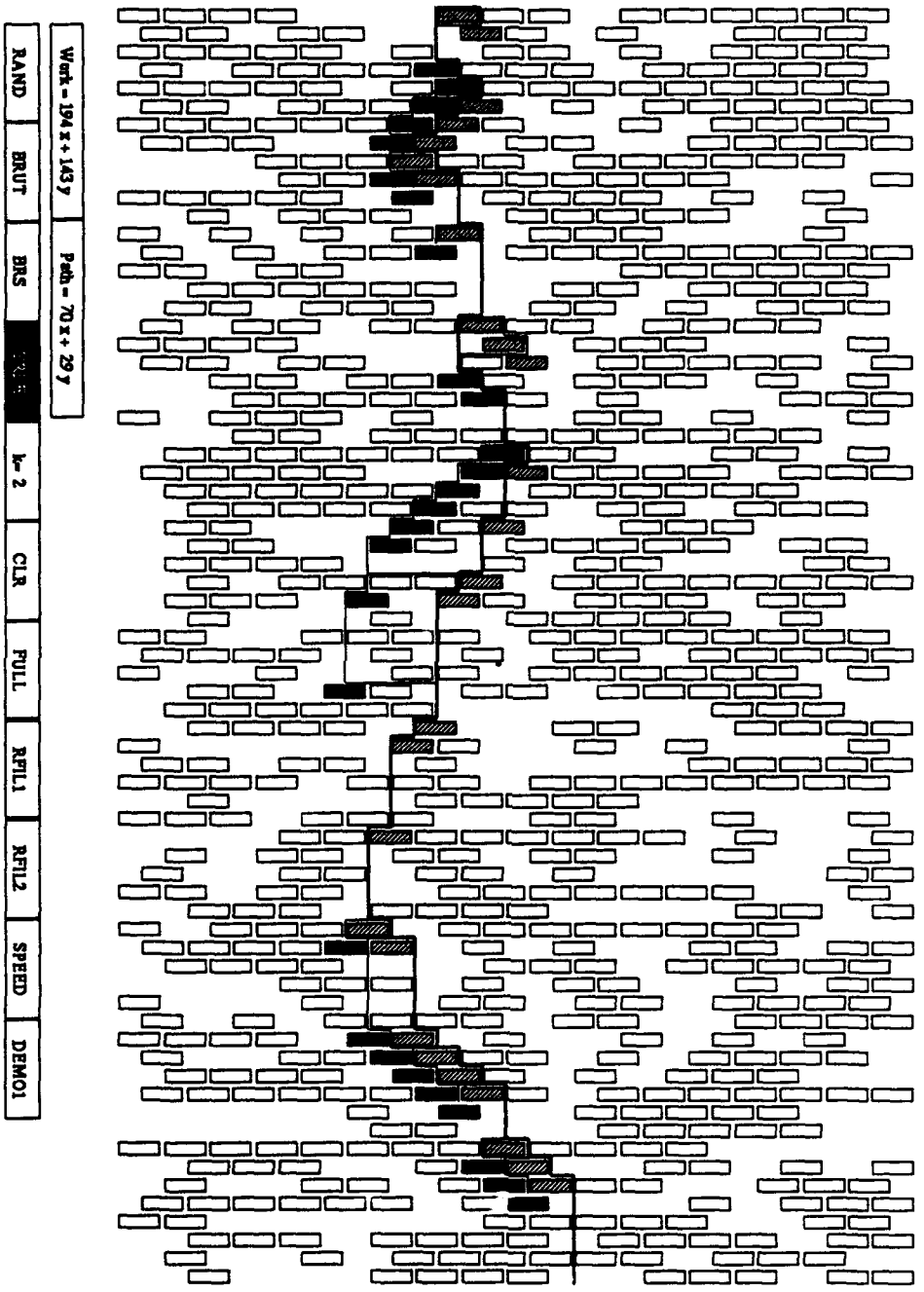

Figure 3.9: Initial search trip, $k = 2$.

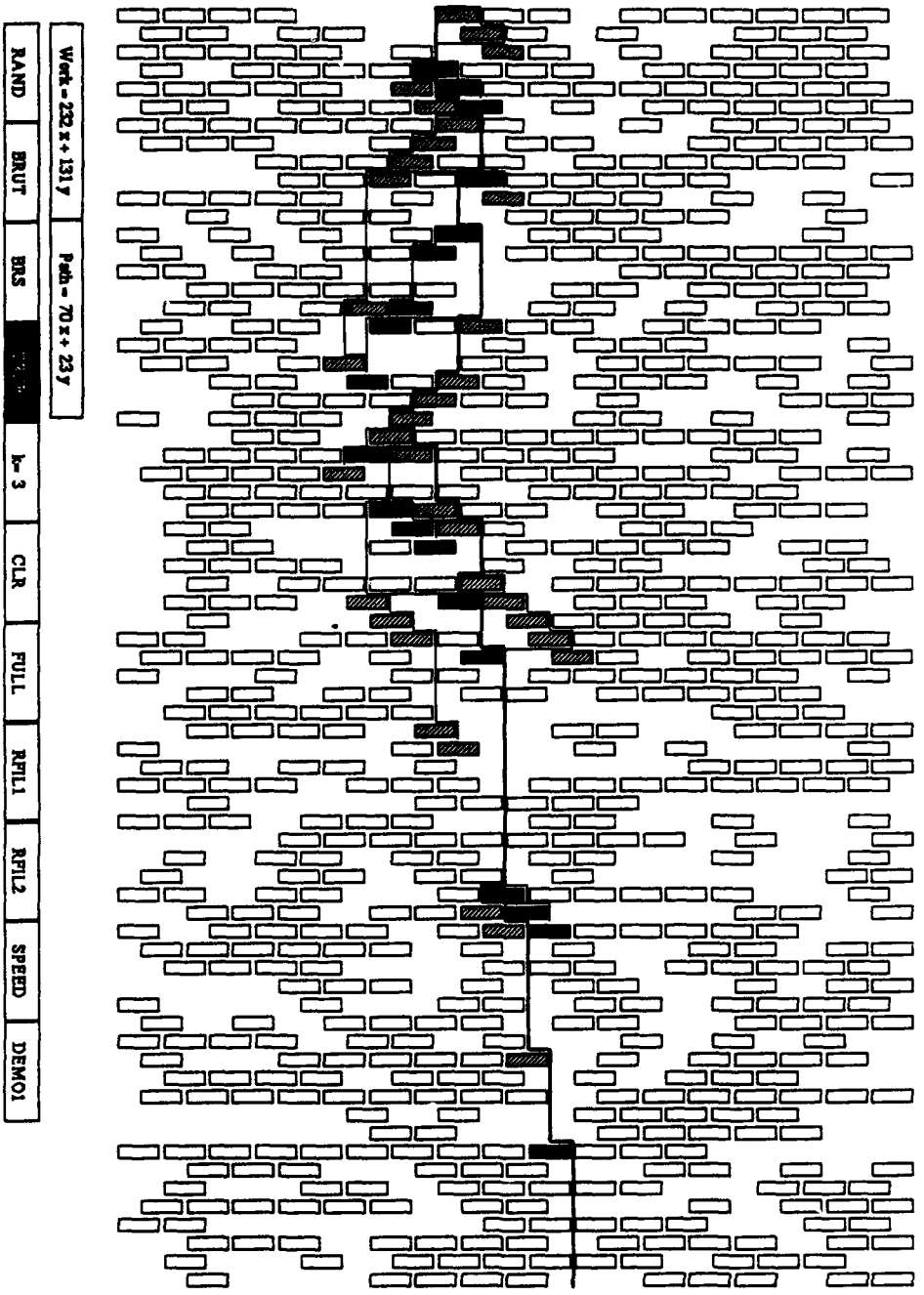

Figure 3.10: Initial search trip, $k = 3$.

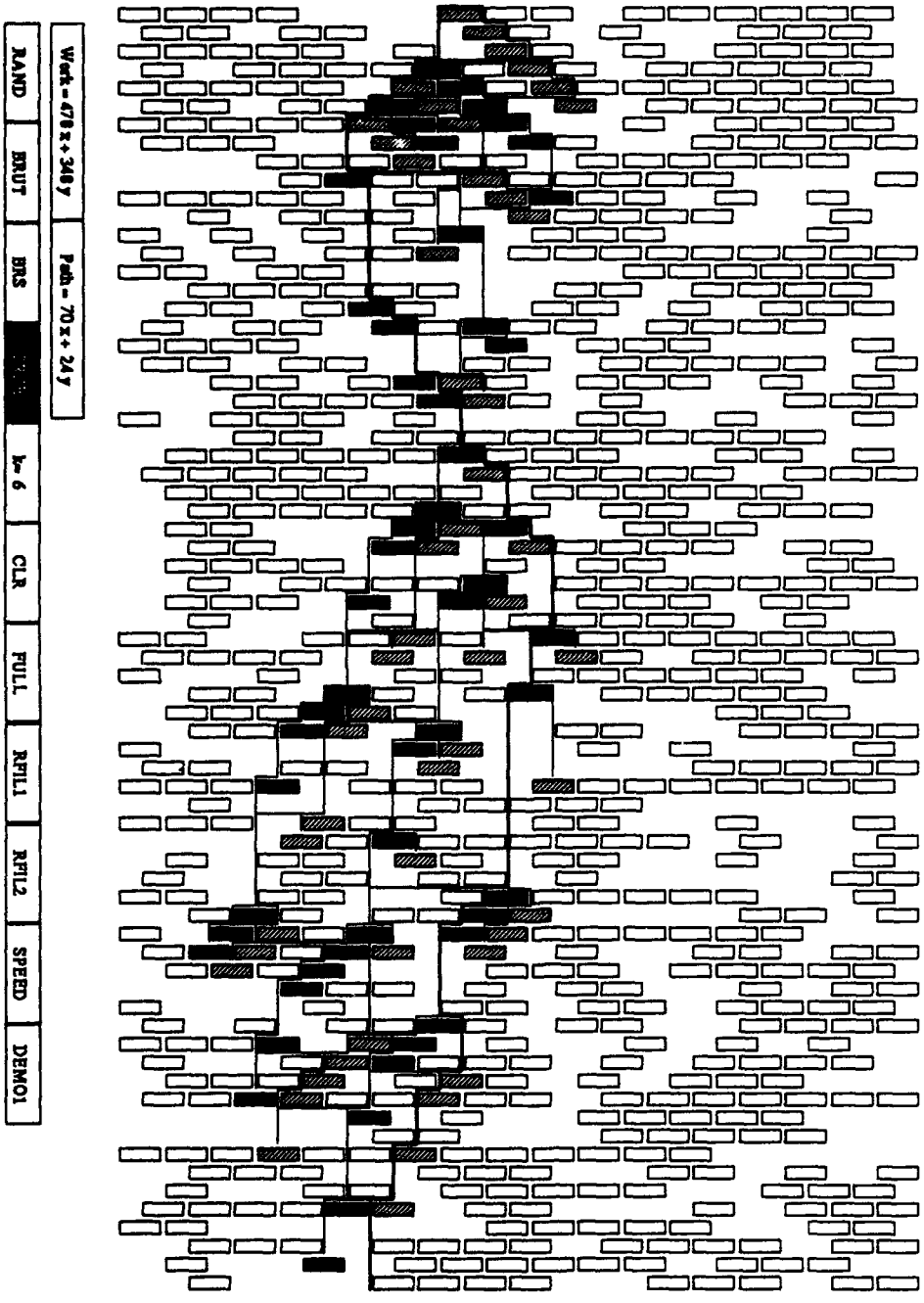

Figure 3.11: Initial search trip, $k = 6$.

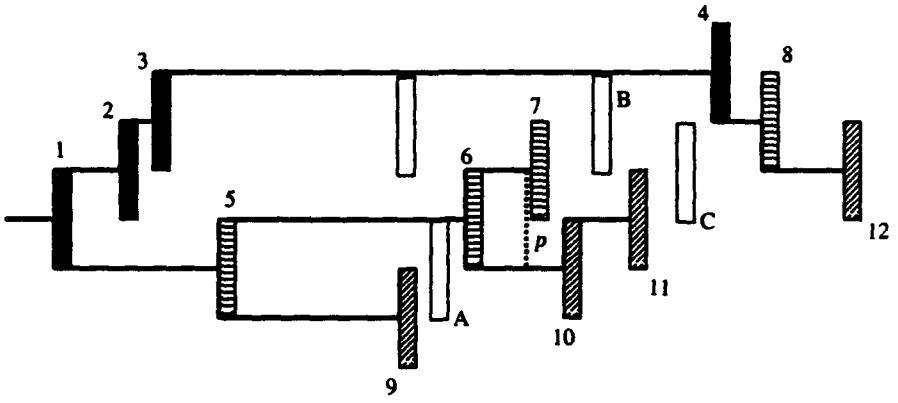

Figure 3.12: A scene showing a 3×4 fence-tree, where the root $F_1(1)$ is obstacle 1. The shaded obstacles are the nodes of the tree, and the dark lines are the edges. Note that the tree defines 3 fences with 4 obstacles each. Differently-shaded obstacles constitute different fences. Other obstacles are not part of any fence.

and whose edges are "short" paths between the nodes. These nodes will constitute the desired collection of fences, and the path from the root node to the rightmost node is the desired $O(L)$ length path that crosses the collection. Furthermore, traversing all edges of this tree with a cost of $O(kL)$ is equivalent to building the desired collection of fences.

The nodes of this tree will be denoted by $F_i(m)$, for $i = 1, 2, \ldots, G$ and $m = 1, 2, \ldots, M$ (for some M to be determined later). The reason for this notation is that $F_i(m)$ will turn out to be the mth obstacle of fence F_i. In a given scene, once we fix an obstacle $F_1(1)$, a unique $G \times M$ *fence-tree* can be defined as follows. The root of this tree is $F_1(1)$. (For illustration we refer the reader to Fig. 3.12 where we show a 3×4 fence-tree.) For brevity we will just say "obstacle P" to mean the center of the left side of obstacle P. The coordinates of obstacle $F_i(m)$ are denoted by $(X_i(m), Y_i(m))$. We say an obstacle Q is *down-right* (*up-right*) from an obstacle P if Q is the first obstacle hit when moving to the right from the *bottom* (*top*) of P. The following rules then define the $G \times M$ fence-tree with root $F_1(1)$ (examples of the rules are pointed out in Fig. 3.12).

Fence-Tree Rules.

1. For $i = 2, 3, \ldots, G$, $F_i(1)$ is down-right from $F_{i-1}(1)$.
   This defines nodes 5,9 in Fig. 3.12.
2. For $m = 2, 3, \ldots, M$, $F_1(m)$ is up-right from $F_1(m-1)$.
   This defines nodes 2,3.
3. For $i = 2, 3, \ldots, G$ and $m = 2, 3, \ldots, M$,
   If $F_i(m-1)$ is to the right of $F_{i-1}(m)$, i.e., $X_i(m-1) \geq X_{i-1}(m)$,
   then $F_i(m)$ is up-right from $F_i(m-1)$
   else $F_i(m)$ is down-right from $F_{i-1}(m)$.
   Node 7 is up-right from node 6, and node 8 is down-right from node 4.

Thus each node $F_i(m)$ (except the root node $F_1(1)$) is defined to be either up-right from $F_i(m-1)$ or down-right from $F_{i-1}(m)$. We can think of the "defining obstacle" of $F_i(m)$ as its (unique) *parent,* and we can think of the "short" (vertical cost h) up-right or down-right path from the parent to $F_i(m)$ as an *edge*. In particular for $i > 1, m > 1$, $F_i(m)$ can only be defined after both its potential parents $F_{i-1}(m)$ and $F_i(m-1)$ have been found, [7] and by Rule 3 its parent is $F_i(m-1)$ if $X_i(m-1) \geq X_{i-1}(m)$ and $F_{i-1}(m)$ otherwise.

This defines a natural *binary rooted tree* structure. The tree structure is visually apparent in Fig. 3.12. The $G \times M$ fence-tree in fact defines exactly the group of fences that we wanted to build:

Theorem 2 (Fence-Tree) *In a given simple scene, for a fixed obstacle P, for a given G and M, the 3 rules above define a unique $G \times M$ fence tree with root P. Moreover, the obstacles in this tree form G disjoint up-fences with M obstacles each, with the first obstacle of the leftmost fence at P. If $M = \lceil \frac{2L}{h} \rceil + \lceil \frac{k}{a} \rceil$ then each of these fences extends across (and possibly beyond) a window of height $2L$ with the root P at the bottom of the window.*

Proof: The uniqueness of the tree follows from the fact that the root $F_1(1)$ is fixed and the three rules define each node $F_i(m)$ uniquely. It is easy to see that the obstacles $F_1(m)$ defined by Rule 2 constitute a fence F_1. Rule 1 defines the initial obstacles of each remaining fence F_i to be down-right from the initial obstacle of the fence F_{i-1} above it. It is clear that for each $i > 1$, the obstacles $F_i(m)$, $m = 1, 2, \ldots, M$

[7] This is not strictly true; it is conceivable for instance that after defining $F_i(m-1)$, one can infer from the positions of the other nodes that $X_{i-1}(m) < X_i(m-1)$ without actually finding $F_{i-1}(m)$. Then $F_i(m)$ can safely be defined to be up-right from $F_i(m-1)$. However we know of no efficient algorithm that takes exploits such a possibility. Our algorithms always find both potential parents of $F_i(m)$ before finding $F_i(m)$.

constitute a fence. We claim that these fences are disjoint. Since the initial obstacle of each fence is down-right from the initial obstacle of the fence F_{i-1} above it, the fences will be disjoint if and only if for each $m = 2, 3, \ldots, M$, $F_i(m)$ is to the right of $F_{i-1}(m)$, and this is guaranteed by Rule 3. For instance, since $F_1(4)$ (obstacle 4) is to the right of $F_2(3)$ (obstacle 7), Rule 3 defines $F_2(4)$ to be down-right from $F_1(4)$. If instead $F_2(4)$ were defined to be up-right from $F_2(3)$, then $F_2(4)$ would be obstacle B, which is left of $F_1(4)$, and fences 1 and 2 would no longer be disjoint.

It is easy to verify that the value $M = \lceil \frac{2L}{h} \rceil + \lceil \frac{k}{a} \rceil$ is just sufficient to ensure that even the fence F_G that starts $Gh = h\lceil \frac{k}{a} \rceil$ below the bottom of the window extends at least $2L$ above the obstacle P (which is $F_1(1)$). [8] ■

Recall that we wanted to define the fence group so that there is a cheap ($O(L)$ length) path that crosses the group. In fact, the unique path in the tree from the root $F_1(1)$ to each node has length at most $O(L)$, as we show below.

Lemma 3 *In a $G \times M$ fence-tree in a simple scene, where $G = \lceil \frac{k}{a} \rceil$ and $M = \lceil \frac{k}{a} \rceil + \lceil \frac{2L}{h} \rceil$,*
(a) the unique path in the tree from the root to each node has length $O(L)$, and
(b) the total length of all edges is $O(kL)$.

Proof: Recall first that $k \leq n$ and $L \geq n$. Since the unique tree path from the root to any given node always goes to the right, the total horizontal cost of this path is at most n. On this path, each down-edge leads to a lower fence (and there are only $G = \lceil \frac{k}{a} \rceil$ fences), and each up-edge leads to an obstacle on the same fence (and there are only $M = \lceil \frac{k}{a} \rceil + \lceil \frac{2L}{h} \rceil$ obstacles per fence), the total vertical cost of this path is at most $h(G + M)$. Thus the total length of this path is at most $n + h(G + M) = n + h(2\lceil \frac{k}{a} \rceil + \lceil \frac{2L}{h} \rceil)$, which is $O(L)$ since $h = aL/\sqrt{nk} \leq aL/k$.

To bound the total length of all edges, note that each edge can be associated with a unique node, namely the one on its right. For a given node, the horizontal portion of the edge whose right end is at that node has length at most the horizontal distance between the node and its predecessor on the same fence. Since the obstacles of a given fence are at distinct x-coordinates, the total horizontal portions of the edges associated with a given fence add up to at most $n \leq L$. The total length of the vertical portions of these M edges is clearly $hM = O(L)$. The desired bound then follows by summing over all $G \leq k$ fences. ■

Thus if the robot traverses all edges of this tree it will have found not only a group of disjoint fences but also a cheap path that crosses all of them.

[8] The fences don't necessarily have to start exactly at the bottom of the window; the important thing is that they extend at least across the window

Note incidentally that there are GM obstacles in the fence-tree, and $GMh \leq kMh = O(kL)$. Thus we would like the robot to traverse the fence-tree with a cost proportional to h times the number of obstacles in the fences, or a cost proportional to the total length of the tree edges. We remark here that the fence-tree must be traversed online; a simple approach based on depth-first-traversal may not be efficient in general since the algorithm does not know where exactly the nodes are: the robot can locate $F_i(m)$ only after it has located both its potential parents $F_{i-1}(m)$ and $F_i(m-1)$ or at least after it has determined whether or not $X_i(m-1) \geq X_{i-1}(m)$ (for $i, m > 1$).

3.5.5 Traversing the Fence-tree

Conservative Strategies and Jumps

Our algorithm will be a *conservative* tree-traversal strategy in the following sense. It adds a new edge to the current partial tree only when such an edge is certain to be part of the final tree being built. In addition, the algorithm visits a node $F_i(m)$, $i > 1, m > 1$, only after both its possible parents $F_{i-1}(m)$ and $F_i(m-1)$ have been visited.

At any given time during a (conservative) tree traversal, the robot either *adds a new edge* from the current node, or *jumps* from the current node to a different (already-discovered) node in the partial tree. Recall that an efficient strategy is one where the total cost is at most a constant times either (a) the total length of the tree edges, or (b) total heights of all the obstacles in the tree (i.e. h times the number of obstacles). So to show that a given conservative strategy is efficient it suffices to argue that the total jump cost is bounded by one of these two quantities. We do not know of any efficient strategy that jumps by walking only along tree edges. Thus our algorithm occasionally moves outside the tree when performing a jump.

A Good Traversal Order is Crucial

We must point out here that even when the restriction to move along tree edges is removed, it is still not the case that just any node-visiting order will be efficient. In particular suppose we try to build the fence-tree of Fig. 3.6 by visiting the nodes in fence-order, i.e., we first visit all nodes on the top fence, then visit all nodes of the second fence, and so on. With this node-visiting order, there is a jump from every node of the second and lower fences; every time an obstacle on one of these fences is discovered by adding a new down-right edge, the algorithm jumps to the fence above in order to find the next obstacle on the fence. Examples of jumps are 2 to 6, 7 to 11, 12 to 16, 3 to 7, 8 to 12, etc. By adding enough additional obstacles in the scene, one can ensure that the shortest path in the scene for each jump is almost as long as

the path in the tree that connects the nodes. Thus we can imagine "charging" each jump to the edges in the corresponding tree path. The problem now is that the tree edges get charged more than a constant number of times. For instance, the 3-to-7 jump is charged to the edges ending at obstacles 2,3,6,7; the 4-to-8 jump is charged to the edges ending at 2,3,4,6,7,8. Thus each jump from the initial obstacle P of a fence charges the series of down-right edges in the tree from obstacle 1 down to P. This is clearly too expensive.

We saw before that the cheapest way to traverse a fence-tree of the form shown in Fig. 3.6 is to build the fences "concurrently" rather than "sequentially", i.e. first visit the initial obstacles of each fence (by down-right edges), then visit the second obstacle of each fence, and so on. Although this strategy is efficient for the particular kind of fence tree shown in the figure, it may not be efficient in general: A scene can be constructed along the lines of the scene of Fig. 3.5 that forces a "concurrent" algorithm to travel a large distance when building the fence-tree. Thus in general neither a purely sequential nor a purely concurrent node-visiting order is efficient. It turns out that a combination of these extremes produces an efficient traversal order.

The key problems in designing a traversal strategy therefore are (a) deciding the order in which the nodes will be discovered, and (b) designing the jump procedures. To aid in the analysis, our jump procedures are designed in such a way that portions of the walk can be charged to either the heights of some obstacles (either already discovered, or to be discovered) in the tree, or to the tree edges. Our node visiting order ensures that different jumps are charged to different parts of the tree, so that each edge and obstacle in the tree is charged at most a constant number of times.

Designing the Traversal Algorithm

Let us first note that during any conservative tree traversal, the partial tree contains some initial prefix (i.e. the leftmost i obstacles, for some i) of each fence. In addition, the following property always holds:

> **Progress-Ordering** Invariant: No fence has more obstacles than a fence above it (since $F_i(m)$, $i > 1$, can be visited only after visiting $F_{i-1}(m)$).

We will now describe the traversal algorithm for a fence-tree corresponding to a group of G *up*-fences with M obstacles each. Rather than present all the details of the algorithm right away, we will motivate intuitively the design of the algorithm.

A key invariant maintained by our algorithm is the following:

> **Last-Node** Invariant: At any time, a new edge is only added from the *last* (highest) node of a partial fence.

Notice that this invariant is *not* satisfied by a strategy that builds the fences one by one sequentially (as described previously). This invariant implies that the jumps made by our strategy are always from the last node of one fence to the last node of another. In fact in our algorithm the jumps are always either to the fence below or the fence above the current fence. Moreover, when the algorithm jumps to the fence above, it simply retraces the path that was last used to jump down to the current fence from the fence above. Thus we need only design the procedure for downward jumps. Intuitively, the fact that jumps are always between the last nodes of the fences ensures that only "recent" portions of the tree are charged for a jump, thus avoiding over-charging any one portion of the tree.

In the remainder of this chapter, we will often identify a fence with its last obstacle; thus when we say "fence F_i is left of obstacle P" we mean that the last obstacle of F_i is left of P. We use (X_i, Y_i) to denote the coordinates of (the center of the left side of) the last obstacle of F_i, and M_i will denote the number of obstacles currently in F_i. We will find it convenient to associate an edge of a fence-tree with the obstacle at its right end, and we define the coordinates of an edge to be the coordinates of its associated obstacle.

At any stage the algorithm either (a) *makes progress* on the current fence, i.e., adds a new up-right edge from the current node, (b) goes down to the next lower fence, either by means of a *down-right edge,* or by *jumping* to the next lower fence, or (c) *returns* to the fence above the current fence. To describe the algorithm we need to specify when each of these actions must be taken.

Firstly, note that in order to maintain the **Last-Node** invariant, the algorithm must not add an obstacle to the current fence (by an up-right edge) if there is a *possibility* of a new down-right edge from the current node. Such a possibility exists if and only if the current node, say $F_i(m)$, is to the right of the last obstacle, say $F_{i+1}(p)$, of the fence below, i.e., $X_i(m) > X_{i+1}(p)$. To see why, note that if $F_i(m)$ is to the right of $F_{i+1}(p)$, then $p < m$. Thus there is a possibility that when $F_{i+1}(m-1)$ is eventually found (if it hasn't yet been found), it might be to the *left* of $F_i(m)$, so there would be a down-right edge from $F_i(m)$. On the other hand, if $X_i(m) \leq X_{i+1}(p)$, then there cannot be any new down-right edge from $F_i(m)$.

Therefore let us insist that the robot always goes down from the current node $F_i(m)$ to the next lower fence F_{i+1} whenever $F_i(m)$ is to the right of F_{i+1}. There are two ways the robot could move to F_{i+1}. If F_{i+1} has $m-1$ obstacles, this means there is a down-right edge from $F_i(m)$, and the algorithm adds this edge. Otherwise, the algorithm jumps down to the last obstacle of F_{i+1} (in a manner to be specified later). Thus a first cut at the fence-tree traversal algorithm might look like this (we assume the algorithm starts at a given obstacle $F_1(1)$, and an empty fence is considered to have its "last" obstacle at x-coordinate $-\infty$):

Repeat until all $G \times M$ nodes on the tree have been found:
(Let $F_i(m)$ be the current node).

1. **If** the current node $F_i(m)$ is to the right of the last obstacle of the fence F_{i+1} below, **then** go down to F_{i+1}, either by adding a down-right edge (if F_{i+1} has $m-1$ obstacles), or by jumping to F_{i+1} using procedure **JumpDownLeft**.
2. **Else** if it is legal (i.e. either $i = 1$ and $F_{i-1}(m+1)$ has been found and $X_{i-1}(m+1) \leq X_i(m)$), add an up-right edge.
3. **Else** jump to the fence F_{i-1} above the current fence.

Observe incidentally that since the robot always moves down to the fence F_{i+1} below when the current fence F_i is ahead of F_{i+1}, the following invariants must hold at all times:

Ordering Invariant: All fences below the current fence F_i are x-ordered, i.e., for all $j > i$, $X_j \leq X_{j+1}$.

Almost-Ordering Invariant: No fence has more than one obstacle to the right of a lower fence.

In order to fully specify the above algorithm, we need to describe the procedure **JumpDownLeft**, which we do below. As we will soon see, this algorithm does not quite satisfy our desired bounds. However, we will be able to repair it by the addition of one more procedure.

Procedure JumpDownLeft: to jump from $F_u(m)$ to $F_{u+1}(p)$. (see Fig. 3.13). Necessary conditions for a call to this procedure: $X_u(m) > X_{u+1}(p)$ and $p < m-1$.

A. Starting at $F_u(m)$, move left along the tree edges until exactly above the last obstacle P of F_{u+1}, i.e. the x-coordinate equals X_{u+1}. (A to B)

B. Move vertically down to P. (B to C)

Let us consider how we might charge the cost of this jump (See Fig. 3.13). From the **Almost-Ordering** Invariant it follows that the set E_m of tree edges retraced (even partially) in phase A consists of zero or more down-right edges, followed by at most one up-right edge. Therefore the edges in E_m have their right ends on obstacles $F_u(m)$, $F_{u-1}(m)$, ...; let S_m be the set of these obstacles. Let T_{u+1} denote the set of *new* obstacles that will be added to F_{u+1} after this jump, up to $F_{u+1}(m-1)$. From the figure it is easy to see that the cost of phase A equals the length of the edges E_m,

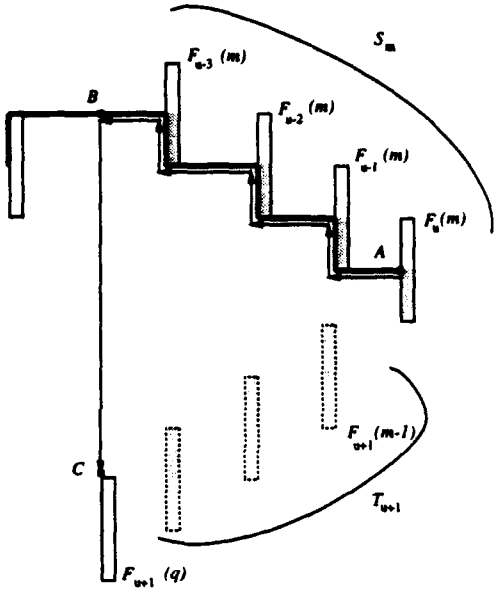

Figure 3.13: Showing a use of procedure **JumpDownLeft** to jump from $F_u(m)$ to (the last obstacle of) F_{u+1}. Solid-boundary rectangles are obstacles found so far in the tree. Rectangles with dotted boundaries are nodes that will be eventually found in the tree. Thick solid lines are tree edges. The thin solid line is the path followed when executing the procedure. The procedure starts from A (obstacle $F_u(m)$), retraces tree edges to the left to point B, then goes vertically down to C, at the top of the final obstacle of F_{u+1}. The set E_m is the set of edges retraced in AB. The length of BC exactly equals the total height of the ***shaded halves*** of the obstacles in $S_m \cup T_{u+1}$.

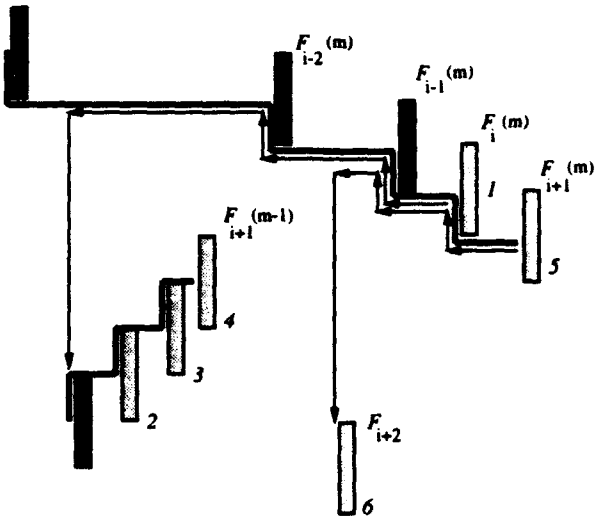

Figure 3.14: How with the simple algorithm, two different uses of **JumpDownLeft** (from $F_i(m)$ to F_{i+1}, and from $F_{i+1}(m)$ to F_{i+2}) can charge the same edges of the fence tree. Obstacles are numbered in the order in which they are *discovered*. Dark lines are tree edges; arrow-lines show the two jump paths.

and the cost of phase B equals $h|S_m \cup T_{u+1}|$. We therefore charge the cost of this use of **JumpDownLeft** to the edges E_m and the obstacles $S_m \cup T_{u+1}$.

Unfortunately, with the above simple algorithm, some of the edges E_m charged in this jump may be charged again by another use of **JumpDownLeft**. (See Fig. 3.14) Indeed, suppose the robot jumps from $F_i(m)$ down to F_{i+1} and eventually arrives at $F_{i+1}(m-1)$. Suppose $X_{i+1}(m-1) < X_{i+2} < X_i(m)$, and $M_{i+2} < m-1$ (this means there will be an up-right edge from F_{i+2}). At this point the robot will return to $F_i(m)$ and then add a down-right edge to $F_{i+1}(m)$. Now step 1 applies again, and since $M_{i+2} < m-1$ the robot executes **JumpDownLeft** to go down to F_{i+2}. Since $X_{i+2} < X_i(m)$, some of the edges retraced in phase A of this procedure will overlap with those in the set E_m that was charged by the $F_i(m)$-to-F_{i+1} jump. It is not hard to see that this is the only sort of situation when two different invocations of **JumpDownLeft** can charge the same portion of the tree; to be precise, no overlap of charging can occur between two calls to **JumpDownLeft** from $F_i(m)$ and $F_j(m')$ when $m \neq m'$.

Fortunately it turns out we could have foreseen this re-charging situation, and taken steps to avoid it. In the above example, when the robot is at $F_{i+1}(m-1)$, it is clear that the *next* obstacle $F_{i+1}(m)$ of F_{i+1} must be to the right of $F_i(m)$ (and therefore to the right of F_{i+2}), and therefore there will be a jump from $F_{i+1}(m)$ to F_{i+2}. This second jump could have been avoided if the robot jumped down from $F_{i+1}(m-1)$ to

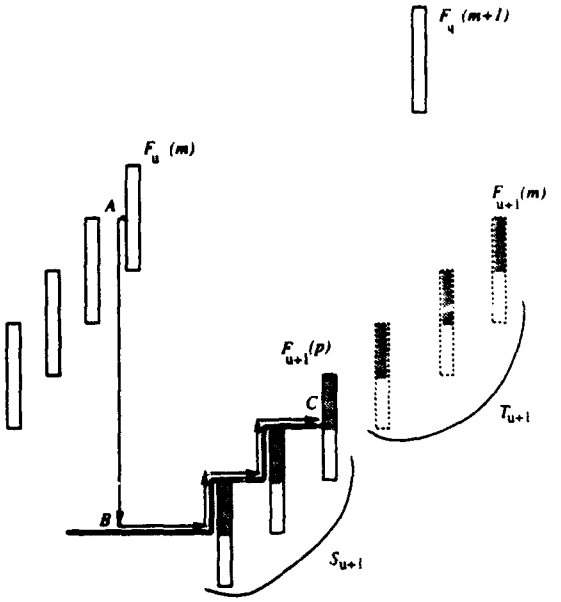

Figure 3.15: Showing a use of procedure **JumpDownRight** to jump from $F_u(m)$ to $F_{u+1}(p)$. Solid-boundary rectangles are obstacles found so far in the tree. Rectangles with dotted boundaries are nodes that will be eventually found in the tree. Thick solid lines are tree edges. The thin solid line shows the path followed when executing the procedure. The procedure starts from A (obstacle $F_u(m)$), goes vertically down to a tree edge (point B), then follows the edges to the right to the final obstacle of F_{u+1} (point C). The set E_{u+1} is the set of edges followed in BC. The length of AB exactly equals the total height of the *shaded halves* of the obstacles in $S_{u+1} \cup T_{u+1}$.

F_{i+2} instead of returning to F_i.

We can generalize from this example and insist that the robot must jump down from $F_i(m)$ to F_{i+1} whenever it becomes clear that (a) the *next* obstacle $F_i(m+1)$ of the current fence F_i will be to the right of F_{i+1}, *and* (b) there will be an up-right edge from F_{i+1}. Condition (a) is equivalent to saying that for some fence F_q above F_i, $F_q(m+1)$ is to the right of F_{i+1}, and condition (b) is just $M_{i+1} < m$. The procedure (call it **JumpDownRight**) to jump down (to the right) from $F_i(m)$ to F_{i+1} is simple (see Fig. 3.15):

Procedure JumpDownRight: to jump from $F_u(m)$ to $F_{u+1}(p)$.
Necessary condition for a call to this procedure: $X_u(m) < X_{u+1}(p)$ and $p < m$.

A. Move vertically down until on a tree edge. (A to B in Fig 3.15)

B. Follow tree edges to the right up to the last obstacle of F_{i+1} (B to C).

It should be clear that the set E_{i+1} of tree edges followed (even partially) in phase B consists of at most one down-right edge followed by zero or more up-right edges. Moreover all these edges have their right ends on obstacles of F_{i+1}. Let S_{i+1} be the set of these obstacles. Let T_{i+1} be the set of *new* obstacles that will be found on F_{i+1}, up to $F_{i+1}(m)$. Clearly the length of phase A equals $h|S_{i+1} \cup T_{i+1}|$, and length of phase B is at most the length of the edges E_{i+1}. We thus charge the cost of **JumpDownRight** to the edges E_{i+1} and the obstacles $S_{i+1} \cup T_{i+1}$.

Another situation (which we overlooked in the above simple version) where the robot must jump down to the next lower fence is if the current fence is complete (i.e. the robot is at $F_i(m)$ and $m = M$).

Our final algorithm is then:

> **Algorithm** FindFenceTree:
> **Repeat until** all $G \times M$ nodes on the tree have been found: (If at any stage the wall $x = n$ is reached, then HALT).
>
> Let $F_i(m)$ be the current obstacle.
>
> 1. **If** $X_i(m) > X_{i+1}$ **then**:
>   if $M_{i+1} = m - 1$ then add a **down-right** edge to $F_{i+1}(m)$,
>   *otherwise* **JumpDownLeft** *to* F_{i+1}.
> 2. **Else if** $M_{i+1} < m$, and either $m = M$, or for some fence F_q above F_i, $X_q(m+1) > X_{i+1}$ **then** **JumpDownRight** to F_{i+1}.
> 3. **Else if** it is legal (i.e. either $i = 1$ or $F_{i-1}(m+1)$ has been found and $X_{i-1}(m+1) \leq X_i(m)$) **then** add an **up-right** edge.
> 4. **Else jump** to the fence F_{i-1} above the current fence.

Note that since we have only increased the set of possible situations when the robot must jump to the next lower fence, all three invariants (**Last-Node**, **Ordering**, **Almost-Ordering**) *continue to hold*. In addition, the following property of the algorithm is easily seen to hold:

> **Jump-Up** Invariant: Whenever the algorithm jumps from a fence F_i to the fence F_{i-1} above, either F_i has the same number of obstacles as F_{i-1}, or it has exactly one less obstacle and its last obstacle is left of the last obstacle of F_{i-1}, i.e. $X_i < X_{i-1}$.

Of course, we now have two different jump procedures and we must make sure that for *each* procedure, different invocations charge different portions of the tree. We prove this below.

In the analysis, we will find it convenient to associate an edge in the tree with the obstacle at its right end, and we define the coordinates of an edge to be the coordinates of its associated obstacle. We say an edge is "on a fence F_i" if its associated

obstacle belongs to F_i. When we say an object A (edge, obstacle, etc) is left (right) of another object B we will mean that the x-coordinate of A is strictly smaller (greater respectively) than that of B.

Lemma 4 *The total cost of all invocations of procedure* **JumpDownLeft** *is* $O(kL)$.

Proof: Consider a particular call to **JumpDownLeft** to jump from $F_u(m)$ to $F_{u+1}(p)$ (see Fig. 3.13).

Clearly the sets E_m and S_m associated with this call will overlap with the corresponding sets of a future call to **JumpDownLeft** from $F_j(m')$ only if $m' = m$. Also, as we shall see below, the set T_{u+1} will not overlap with the corresponding set in any future call to **JumpDownLeft** from $F_u(m')$. Thus it suffices to show that the charged sets E_m, S_m, T_{u+1} associated with this call do not overlap with the corresponding sets of any future call to **JumpDownLeft** from some obstacle $F_j(m)$. Firstly note that this procedure cannot be called later from $F_j(m)$ for any fence F_j above F_u since F_{j+1} has at least m obstacles (by the **Progress-Ordering** invariant) and therefore is to the right of $F_j(m)$, which means the necessary conditions for using this procedure would not be satisfied.

We claim that when the robot eventually jumps back from F_{u+1} to $F_u(m)$ after the current jump, F_{u+1} and every fence below and left of F_u will have at least $m-1$ obstacles. The necessary condition $p < m-1$ for this procedure's invocation then implies that no fence below and left of $F_u(m)$ can be the destination of a future call to **JumpDownLeft** from the mth obstacle of any fence. It then follows that none of the elements of E_m and S_m will be charged again (since none of the elements of E_m and S_m are right of $F_u(m)$ and none of the elements of the E-set and S-set in any call to **JumpDownLeft** are left of the destination of the call). Another implication of this claim is that the charged set T_{u+1} of *undiscovered* obstacles of F_{u+1} will have been discovered before the next call to **JumpDownLeft** from $F_j(m')$ for either $j = u$ or $m' = m$ and so will not be charged again.

We now argue the above claim. That F_{u+1} will have at least $m-1$ obstacles follows from the **Jump-Up** invariant. To prove the rest of the claim, suppose that F_v is the highest fence below F_{u+1} that is left of F_u and has fewer than $m-1$ obstacles. By the **Ordering** Invariant this means that each of the fences between (and including) F_{u+1} and F_{v-1} each have exactly $m-1$ obstacles and are left of F_v (and therefore left of F_u). However when $F_{v-1}(m-1)$ was discovered, step 2 would have applied and the robot would have used **JumpDownRight** to jump to F_v, and it would only return to F_{v-1} when F_v has at least $m-1$ obstacles (by the **Jump-Up** invariant). This is a contradiction, and the claim follows. ■

Lemma 5 *The total cost of all invocations of procedure* **JumpDownRight** *is* $O(kL)$.

Proof: Consider a particular call of **JumpDownRight** to jump from $F_u(m)$ to $F_{u+1}(p)$ (see Fig. 3.15), and let the associated charged sets be $E_{u+1}, S_{u+1}, T_{u+1}$.

It suffices to show that for any future call to this procedure from $F_u(m')$ on the same fence F_u, the charge sets do not overlap with the current charge sets. The **Jump-Up** invariant implies that when the robot jumps back to F_u, F_{u+1} must have at least m obstacles (which implies that **JumpDownRight** cannot be called again from $F_u(m)$, since the necessary condition $p < m$ for using this procedure will not be satisfied). This means that the charged set T_{u+1} of *undiscovered* obstacles will have been discovered before any future call from F_u, and so will not overlap with a future T-set. Furthermore, from the previous discussion of this procedure we know that the *next* obstacle $F_u(m+1)$ of F_u is to the right of F_{u+1}. Since any future call to **JumpDownRight** from F_u can only occur from $F_u(m+1)$ or an obstacle to its right, the sets E_{u+1} and S_{u+1} in the current call will not overlap with the corresponding sets in any future call from F_u (since none of the elements of E_{u+1} and S_{u+1} are to the right of $F_u(m+1)$, and none of the elements of the E-set and S-set in any use of **JumpDownRight** are left of the start point of the jump). This completes the proof. ■

The above two lemmas imply our main theorem:

Theorem 6 *For $G = \lceil \frac{k}{a} \rceil$ and $M = \lceil \frac{k}{a} \rceil + \lceil \frac{2L}{h} \rceil$, the above algorithm for finding a $G \times M$ fence-tree in a simple scene has total cost $O(kL)$.*

3.6 Extension to Arbitrary Axis-Parallel Rectangular Obstacles

We now show how to extend the search algorithm to scenes with arbitrary axis-parallel rectangular obstacles (for brevity we call such scenes *general* scenes). That is, we will show how to explore a distance of $O(L\sqrt{nk})$ and find a path of length $O(L\sqrt{n/k})$. Fortunately it turns out that algorithm **FindFenceTree**, interpreted appropriately, can be used unchanged for these scenes. However, the procedures **JumpDownRight** and **JumpDownLeft** must be modified since for general scenes vertical motion is no longer unobstructed. In fact if all obstacles have width 1 (but arbitrary heights and positions) then even these procedures remain unchanged.

Without loss of generality we will assume throughout that all obstacles have their left sides at integer x-coordinates.

3.6.1 τ-Posts

Throughout this section we will denote the value $L/\sqrt{nk}$ by τ. Recall that in a simple scene if the obstacles have height less than $2\tau = 2L/\sqrt{nk}$ then they can be considered "small" – in the sense that the simple strategy of just moving horizontally forward (walking around any obstacles on the way by the shortest route) achieves the optimal ratio of $O(\sqrt{n/k})$ on *each* trip. Similarly in a general scene if when moving horizontally along some line $y = y_0$, the robot encounters an obstacle (at point P, say) whose top or bottom extends less than τ above or below P, the obstacle can be considered small; the robot can get around the obstacle and return to $y = y_0$ paying a cost of less than $2\tau + 1$. If however both the top and bottom of the obstacle are at least τ away from the point of encounter P, then the obstacle is "big". We then say that there is a *τ-post* at point P, where we define a τ-post as a height-2τ portion of the left side of an obstacle. When we say a τ-post is at point P, we mean its center is at P.

3.6.2 τ-Fences

In a manner analogous to simple scenes, we define an up *τ-fence* F as a sequence of τ-posts at points $(X(1),Y(1)),(X(2),Y(2)),\ldots,(X(M),Y(M))$ such that for $m = 1,2,\ldots,M-1$:

$$X(m) \leq X(m+1) \tag{3.1}$$

$$Y(m+1) = Y(m)+\tau \ (= Y(1)+(m-1)\tau) \tag{3.2}$$

A down τ-fence is defined similarly. Note that consecutive τ-posts of a fence may lie on the *same* obstacle, since the inequality 3.1 is not strict. As before we will only be interested in τ-fences that extend at least across a window of height $2L$ centered vertically at s, i.e., $Y(1) \leq -L$ and $Y(M) \geq +L$. Also, the mth post (counting from the left) of fence F_i is denoted by $F_i(m)$ and its coordinates are $(X_i(m),Y_i(m))$. In Fig. 3.16, the sequence of τ-posts $\langle P_1^1, P_1^2, P_1^3, P_1^4\rangle$ form a τ-fence F_1.

Between the top half of one τ-post and the bottom half of the next τ-post in a τ-fence, there is an axis-parallel rectangular region of height τ, which we call a *band*. A fence is thus a contiguous sequence of bands extending across the window. Note that the band between two consecutive posts that are on the same obstacle is empty. Two fences are said to be *disjoint* if their *non-empty* bands do not overlap. Thus the three fences in Fig. 3.16 are disjoint.

For future reference we define a (right) *τ-path* as the path of the robot when it moves to the right along some line $y = y_0$ as follows: If it hits an obstacle whose nearest corner is less than τ away, it goes around that corner to the point on the opposite

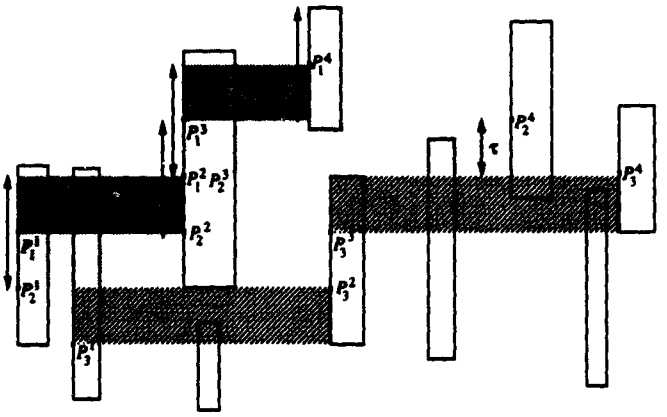

Figure 3.16: A collection of 3 disjoint fences with 4 posts each. The solid rectangles are the obstacles. The bands of different fences are shaded differently. For convenience, post $F_i(m)$ is denoted P_i^m.

side with y-coordinate y_0 and continues to the right: if it hits either a τ-post or the wall, it stops. For instance in Fig. 3.17, the path from A to τ-post $F_1(2)$ is a τ-path. Observe that a τ-path has vertical motion at most 2τ at every (integer) x-coordinate on the path, so:

Fact 1 *A τ-path between two points (x, y) and $(x + \delta x, y)$ has length at most $\delta x + 2\tau\ \delta x$.*

The definition of a point being *left* or *right* of a fence, and of a path *crossing* a fence remain the same as before, except that we use τ-post wherever the word "obstacle" was used previously. Thus a τ-fence costs at least τ to cross, and a collection of k disjoint τ-fences costs at least $k\tau$ to cross.

3.6.3 The Initial Search Trip

Roughly speaking, a general scene is treated as if it is a simple scene with obstacles of height $2h = 2\tau = 2L/\sqrt{nk}$. Recall that for simple scenes, the initial trip consists of building groups of G fences of M obstacles each (where $G = \lceil \frac{k\tau}{h} \rceil$ and $M = \lceil \frac{k\tau}{h} \rceil + \lceil \frac{2L}{h} \rceil$), where each group must be built "cheaply" (i.e. with cost $O(kL)$) and must have a known "short" (cost $O(L)$) path crossing it. Analogously for general scenes we have $h = \tau$ and we would like to build groups k τ-fences with $M = k + \lceil \frac{2L}{\tau} \rceil$ τ-posts each. We will now pay more attention to our progress in the x direction and will build each group with cost at most $O(kL + k\tau\ \Delta x)$, and give each group a group-crossing path of length $O(L + \tau\ \Delta x)$. Here Δx is the x-distance between the leftmost τ-post $F_1(1)$ and right-most τ-post $F_k(M)$ in the group.

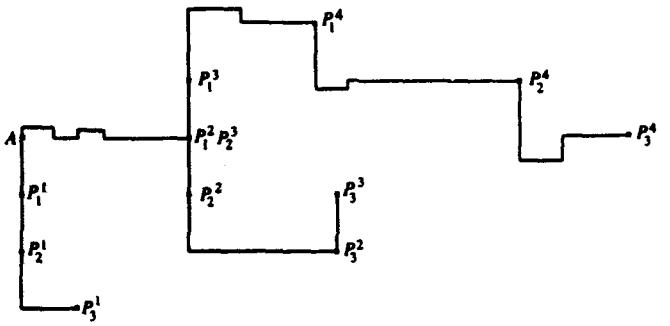

Figure 3.17: A 3×4 τ-fence-tree. The shaded rectangles are the obstacles, and solid lines are tree edges. The fences corresponding to this tree are shown in Fig. 3.16. For convenience, post $F_i(m)$ is denoted P_i^m.

These bounds are sufficient for our purposes for the following reason. A fence costs $\tau = L/\sqrt{nk}$ to cross, there can be at most $\sqrt{nk}$ disjoint τ-fences in the window between s and t, so the algorithm will find at most $\sqrt{n/k}$ groups of k fences each. Since the x-motions do not overlap between the groups of fences, the Δx terms add to at most n, so the total distance traveled is $O(kL\sqrt{n/k} + nk\tau) = O(L\sqrt{nk})$. In addition, the concatenation of the group-crossing paths has total cost $O(L\sqrt{n/k} + n\tau) = O(L\sqrt{n/k})$.

3.6.4 Extending FindFenceTree to General Scenes

Once we fix the τ-post $F_1(1)$ in a scene, the three fence-tree definition rules given for simple scenes can be used in a general scene to define a group of $G = k$ τ-fences with M τ-posts each, with the following interpretation. Firstly, "τ-post" replaces the word "obstacle" everywhere. Secondly, an up-right edge from a τ-post $F_i(m)$ is simply a path that goes up to the top of the τ-post, then right along a τ-path until a τ-post is reached; this τ-post is $F_i(m+1)$. Similarly, a down-right edge from a τ-post $F_i(m)$ is simply a path that goes down to the bottom of the τ-post, then right along a τ-path until a τ-post is reached; this τ-post is $F_{i+1}(m)$.

Given this interpretation, the extension of Algorithm FindFenceTree to arbitrary axis-parallel rectangular obstacles is very simple: wherever the algorithm calls for adding an up-right (down-right) edge from the current τ-post the robot must move up (down) a distance τ, then move horizontally to the right along a τ-path until it hits a τ-post. Of course, as we indicated earlier, the jump procedures must be changed to handle

arbitrary obstacle widths. The main difficulty here is that it is no longer possible to cheaply move down vertically (in Phase A of **JumpDownRight**, and Phase B of **JumpDownLeft**) since there may be many "fat" obstacles on the way.

It is easy to verify that the four invariants **Last-Node**, **Ordering**, **Almost-Ordering** and **Jump-Up** still hold, so the algorithm does find a $k \times M$ τ-fence-tree if it exists with the given post $F_1(1)$ as root. We must show that this algorithm is still "cheap" in a general scene. Note that if the robot does not encounter any "small" obstacles when adding the various edges, then (assuming the jump procedures are cheap) our previous arguments suffice. We begin with the fairly straightforward argument that the in general the cost of going around "small" obstacles is not too large. To do this, we show the analogue of Lemma 3, namely, that the total cost of the tree edges and the cost of the path from the root to the rightmost node are both within our required bounds.

Lemma 7 *Suppose there is a $k \times M$ τ-fence-tree consisting of fences $F_1, F_2, \ldots, F_k$, with $M = k + \lceil \frac{2L}{\tau} \rceil$, where $\tau = L/\sqrt{nk}$, $k \leq n$ and $L \geq n$. Then:*
(a) The unique path in the tree from $F_1(1)$ to $F_k(M)$ has length at most $4L + 3\tau\Delta x$;
(b) The total length of all the edges in the fence-tree is at most $k(3L + 3\tau\Delta x)$

Proof: Since $L \geq n$, we have $\tau = L/\sqrt{nk} \geq 1$. From $k \leq n$ it follows that $k\tau = kL/\sqrt{nk} \leq L$. This implies $M\tau = 2L + k\tau \leq 3L$.

Part (a). There are exactly $(M + k - 2)$ edges in the tree path from $F_1(1)$ to $F_k(M)$. Since the vertical portion of each edge has length τ, and the τ-path portions of the edges do not overlap in the x-direction, the total length of these edges is at most $(M + k - 2)\tau + 2\tau\Delta x + \Delta x$, which is at most $(4L + 3\tau\Delta x)$ from the inequalities above.

Part (b). Note that we can associate each edge with a unique post, namely the one at the right-end of the edge. For any given post $F_i(m)$ other than $F_1(1)$, the x-distance to its parent is at most the x-distance δx to its predecessor $F_i(m-1)$ on the same fence. So the edge associated with this post has length at most $(\tau + 2\tau\delta x + \delta x)$. The sum of the δx terms over all posts of the fence F_i is the x-distance between the first and last posts of F_i, which is at most Δx. So the total length of the edges associated with the M posts of a fence is at most $(M\tau + 2\tau\Delta x + \Delta x)$, which sums to $k(M\tau + 2\tau\Delta x + \Delta x)$ for k fences. This last expression is at most $k(3L + 3\tau\Delta x)$ from the previous inequalities. ■

Thus, just as in simple scenes, we only need to argue that the total cost of each jump procedure is at most a constant times the total length of the tree edges. In the next two subsections we show how these procedures must be modified to handle arbitrary obstacle widths, and argue that they are not too expensive. As before, our approach

will be to charge various phases of the jump to τ-posts (the total length of these is at most the total length of the edges in the tree) or edges in the tree, and argue that for each jump procedure, no portion of the tree is charged too often for different executions of that procedure.

3.6.5 Modifying JumpDownRight

To describe the modifications, it will be useful to introduce the notion of a *greedy down-left* path: it is a path that repeatedly goes "*down* till it hits an obstacle, then to the *left* corner of the obstacle". Other greedy paths are defined similarly. As in the case of simple scenes, we will find it convenient to associate an edge in the fence-tree with the post at its right end.

For simple scenes, phase A of the **JumpDownRight** procedure consists of moving vertically downward until a tree path is hit. This may not be possible in a general scene since the robot may encounter several wide obstacles. Our modified procedure is then the following, and the worst case path of each phase is shown in Fig. 3.18. (Recall that (x, y) denotes the current coordinates at any stage).

> **Procedure JumpDownRight :** to jump from $F_u(m)$ to $F_{u+1}(p)$.
> Necessary conditions for a call to this procedure: $X_u(m) < X_{u+1}(p)$, and $p \leq m$.
> If called outside **JumpDownLeft**, $p < m$ must hold.
>
> **A1.** Move greedy down-left until $y \leq Y_{u+1}(p) + \tau$. (A to B in Fig. 3.18)
>
> **A2.** Move greedy right-down until a tree path is hit. (B to C)
>
> **B.** Follow tree edges to the right until at $F_{u+1}(p)$. (C to D)

Note that the path in phase A1 is bounded on the left by the τ-posts of F_u. Also, it suffices to continue phase A1 until $y \leq Y_{u+1}(p) + \tau = Y_u(p+1) - \tau$ since this is the y-coordinate of the top of the τ-post to which the robot is jumping; even if the greedy right-down path of phase $A2$ goes only to the right, it will hit this τ-post. In the *worst* case, phase A1 follows the right sides of the τ-posts of F_u up to $F_u(p+1)$, and phase A2 just goes vertically down.

Lemma 8 *The total cost of all calls to* **JumpDownRight** *(excluding calls to this procedure by* **JumpDownLeft***) is at most a constant times the total length of all edges in the fence-tree*

Proof: In the following, let us confine ourselves to calls to **JumpDownRight** that are made outside procedure **JumpDownLeft**. Consider a particular call to **JumpDownRight** to jump from $F_u(m)$ to $F_{u+1}(p)$ (see Fig. 3.18). Let E_{u+1} be the set of edges of F_{u+1} whose x-coordinates are in the interval $[X_u(p+1), X_{u+1}(p)]$, and let S_{u+1} be the set

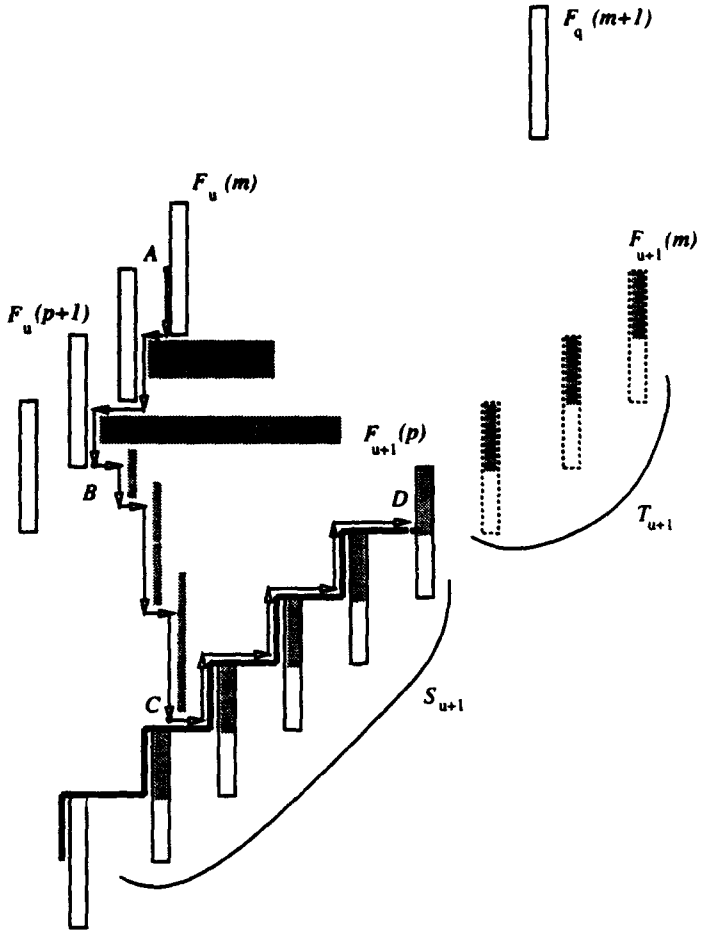

Figure 3.18: A use of the generalized procedure **JumpDownRight** to jump from $F_u(m)$ to $F_{u+1}(p)$, for clarity shown in a scene where the fence posts correspond exactly to obstacles of height 2τ. Shaded rectangles with no boundaries are obstacles that are not part of any fence. Solid-boundary rectangles are nodes found so far in the tree. Dotted-boundary rectangles are nodes that will be eventually found in the tree. Thick solid lines are tree edges. The thin arrow line shows the path followed when executing the procedure.

of τ-posts associated with these edges. Also, let T_{u+1} be the set of *new* τ-posts that will be added to F_{u+1}, up to $F_{u+1}(m)$. Note that

> **Fact 1:** none of the elements of E_{u+1}, S_{u+1} and T_{u+1} are to the left of $F_u(p+1)$,

and from the previous discussion of the procedure **JumpDownRight** we know that

> **Fact 2:** the next post $F_u(m+1)$ of F_u will be to the right of $F_{u+1}(p)$, so none of the elements of E_{u+1} and S_{u+1} are right of $F_u(m+1)$.

It is easy to see that the total *vertical* motion during phases A1 and A2 is at most $\tau|S_{u+1} \cup T_{u+1}|$. The cost of the *horizontal* motion in phases A1 and A2 plus the cost of phase B is at most *twice* the length of the edges in E_{u+1}. Thus, we charge the total cost of this procedure to the edges E_{u+1} and the τ-posts $S_{u+1} \cup T_{u+1}$.

As before it suffices to show that for any future use of this procedure (outside procedure **JumpDownLeft**) to jump from $F_u(m')$ to $F_{u+1}(p')$, the charged sets do not overlap with the corresponding sets in the present call. First, note that by the **Jump-Up** invariant, when the robot jumps back to F_u, F_{u+1} must have at least m posts (which implies that **JumpDownRight** cannot be called again outside procedure **JumpDownLeft** from $F_u(m)$, since this would violate the necessary condition $p' < m'$ for using this procedure outside **JumpDownLeft**). This means that the charged set T_{u+1} of *undiscovered* posts will have been discovered before any future call from F_u, and so will not overlap with a future T-set. This also implies that the destination $F_{u+1}(p')$ of any future **JumpDownRight** from F_u cannot be left of $F_{u+1}(m)$. From Fact 1 above it then follows that none of the elements of the E-set and S-set of this future jump will be to the left of $F_u(m+1)$. Therefore the sets E_{u+1} and S_{u+1} (none of whose elements are right of $F_u(m+1)$, by Fact 2 above) in the current call will not overlap with the corresponding sets in any future call from F_u, which completes the proof. ■

3.6.6 Modifying JumpDownLeft

For simple scenes, phase B of the **JumpDownLeft** procedure consists of moving down vertically till at the top of the destination τ-post. This may not be possible in a general scene since the robot may encounter several wide obstacles. Our modified procedure is then the following, and is illustrated in Fig. 3.19:

> **Procedure JumpDownLeft :** to jump from $F_u(m)$ to $F_{u+1}(p)$.
> Necessary conditions for a call to this procedure: $X_u(m) > X_{u+1}(p)$ and $p < m-1$.

A. Retrace edges to the left until $x = X_{u+1}(p)$. (A to B in Fig. 3.19)

B1. Move greedy down-left until for some j, the robot is left of and below $F_j(m-1)$, or it is below $F_u(m-1)$.
In the former case (B to C), repeatedly **JumpDownRight** to the $(m-1)$st post on the next lower fence until at $F_u(m-1)$. (C to D)

B2. **JumpDownRight** to $F_{u+1}(p)$ (D to E).

Some explanation is in order here. Suppose F_r is the highest fence reached in phase A, i.e. the last edge retraced is on fence F_r. By the **Almost-Ordering** invariant, no fence can have more than one post to the right of a lower one, so all but the last edge retraced in phase A must be down-right edges ending on posts $F_{r+1}(m)$, $F_{r+1}(m), \ldots, F_u(m)$. The same invariant implies that the x-coordinate of the robot at the end of phase A (i.e. $X_{u+1}(p)$) lies in the interval $[X_j(m-1), X_j(m)]$, for each $j = r, r+1, \ldots, u$. This means that in phase B1 when the robot goes greedy down-left it *must* arrive below and left of some $F_j(m-1)$ (for some $r < j < u$) or below $F_u(m-1)$. In the former case, procedure **JumpDownRight** (modified trivially since the robot may not start exactly at a τ-post) can be used to jump down to the post $m-1$ on the next lower fence; and doing this repeatedly will lead to $F_u(m-1)$. Now **JumpDownRight** (again slightly modified since the start point may not exactly be the post $F_u(m-1)$ in case the greedy down-left path in phase B1 leads directly to a point below this post) can be used to jump down to $F_{u+1}(p)$.

Lemma 9 *The total cost of all calls to* **JumpDownLeft** *(including the cost of calls to* **JumpDownRight***) is at most a constant times the total length of all the tree edges.*

Proof: For a particular call to **JumpDownLeft** let E_m, S_m, T_{u+1} be defined as before for simple scenes. The length of phase A is clearly still at most the total length of edges in E_m. Let F_r be as above the highest fence reached in phase A.

Let us consider the cost of phase B1. There are two possibilities: while going greedy down-left the robot may reach a point below $F_u(m-1)$, or it may reach a point left of and below $F_j(m-1)$ for some F_j above F_u. In the former case, the length of this phase is at most $\tau|S_m|$. In the latter case, the robot repeatedly uses **JumpDownRight** to jump to the $(m-1)$st post on the next lower fence until it reaches $F_u(m-1)$. Each use of **JumpDownRight** here is to jump from $F_j(m-1)$ to $F_{j+1}(m-1)$, for $j = r, r+1, \ldots, u-1$. From the analysis of that procedure it is clear that the cost of these invocations of **JumpDownRight** is at most a constant times the lengths of the edges in the E-sets associated with these calls. These are precisely the edges of F_{j+1} whose x-coordinates are in the interval $[X_j(m-1), X_{j+1}(m-1)]$, for $j = r, r+1, \ldots, u-1$. Let us call these edges type-a edges. Thus the total cost of this phase is at most the sum of $\tau|S_m|$ and a constant times the lengths of the type-a edges.

Phase B2 is an execution of procedure **JumpDownRight**, and from the analysis of that procedure it follows that the cost of this phase is at most the sum of $\tau|T_{u+1}|$ and a constant times the lengths of edges of F_{u+1} whose x-coordinates are in the interval $[X_u(m-1), X_{u+1}(p)]$. Let us call these edges type-b edges.

We therefore charge the cost of this jump to S_m, T_{u+1} and the type-a and type-b edges. The important point to note here is that all of the type-a and type-b edges are on the fences F_{r+1}, F_{r+2}, ..., F_u, F_{u+1}, i.e., the *highest* fence that can possibly contain any type-a or type-b edges is the one *below* the highest fence to which the robot backs up in phase A (i.e. fence F_r).

As in the case of simple scenes, it suffices to show that none of these charged portions (posts or edges) are charged again in a future call to **JumpDownLeft** from $F_i(m)$ for any $i > u$. That T_{u+1} will not be charged again can be shown by the same argument as before. By exactly the same argument as before, the destination of any future call to **JumpDownLeft** from $F_i(m)$ cannot be to the left of $F_u(m)$, which implies that the robot will backup to no higher than $F_{u+1}(m)$ in phase A. Thus the S-set and E-set of this future call will not overlap with S_m and E_m respectively. This also implies that the highest fence containing any type-a/type-b edges in this future call is F_{u+2}, which is *below* the *lowest* fence (namely F_{u+1}) containing type-a/type-b edges in the present call. Thus none of the charged elements in the present call will be charged again by a future call to **JumpDownLeft**. ∎

3.7 An incremental algorithm

We describe here an improvement of our cumulative algorithm, so that the per-trip ratio on the i'th trip, for all $i \leq n$, is $O(\sqrt{n/i})$. Let us for simplicity say that we know L. From the earlier results in this paper, we know that by searching a distance at most $cL\sqrt{nk}$ we can find an s-t path of length at most $c'L\sqrt{n/k}$, for some constants c, c' and any $k \leq n$.

Let us suppose that at the end of i trips we know an s-t-path π of length at most $c'L\sqrt{n/i}$. What we now want to do is to search with cost at most $cL\sqrt{n2i}$ and find a path of length at most $c'L\sqrt{n/2i}$. Let us denote by Π the path we would have traveled if we did this entire search in one trip. In order to maintain a per-trip ratio of $O(\sqrt{n/i})$, we spread the work of Π over the next i trips as follows. Each trip consists of two phases: The first is a *search* phase, where we walk an additional portion of Π of length $\frac{1}{i}cL\sqrt{n2i} = cL\sqrt{2n/i}$, starting from where we left off on the previous trip. We can always do this because the fences are in a tree structure, so that the last point in Π during the previous search can always be reached from the start point by a known short path whose length adds only a small constant factor to the total trip

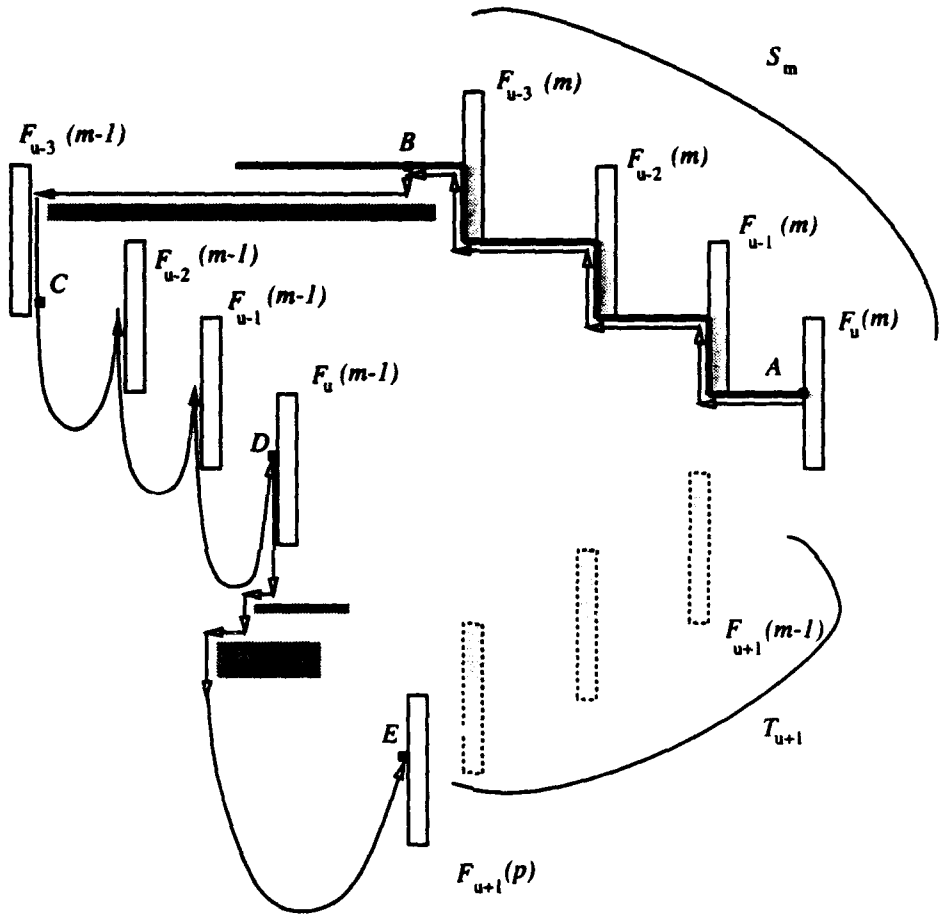

Figure 3.19: A use of procedure **JumpDownLeft** to jump from $F_u(m)$ to $F_{u+1}(p)$, for clarity shown in a scene where the fence posts correspond exactly to obstacles of height 2τ. Shaded rectangles with no boundaries are obstacles that are not part of any fence. Solid-boundary rectangles are nodes found so far in the tree. Dotted-boundary rectangles are nodes that will be eventually found in the tree. Thick solid lines are tree edges. The thin arrow line is the path followed when executing the procedure. Each curved arrow line shows an execution of procedure **JumpDownRight**.

length. Once the search phase is completed, we "give up" and enter the *follow* phase, where we complete the trip by joining (by a greedy path) the known path π of length $c'L\sqrt{n/i}$, and following it to t. Thus our trip length is still $O(L\sqrt{n/i})$. Since in each such search-follow trip we traverse a portion of Π of length $cL\sqrt{2n/i}$, and the length of Π is at most $cL\sqrt{2ni}$, after i trips we will have completely walked the path Π. So after the first $2i$ trips we have a path of length at most $c'L\sqrt{n/2i}$. This reestablishes our invariant. Since initially we can find a path of length $c'L\sqrt{n}$ (to start off the induction), we have the following theorem:

Theorem 10 *There is a deterministic algorithm R that for every $i \leq n$ achieves a per-trip ratio on the i'th trip, $\rho_i(R,n)$, of $O(\sqrt{n/i})$.*

3.8 Modification for Point-to-Point Navigation

Our algorithms can be extended to the case where t is a point rather than a wall, with the same bounds, up to constant factors, as follows. Let us assume for simplicity that the shortest path length L is known. As before, if we do not know L, we can use the standard "guessing and doubling" approach and suffer only a constant factor penalty in performance. On the first trip, the robot can get to t using the optimal point-to-point algorithms of [19] or [13], with a single-trip ratio of $O(\sqrt{n})$. Once at t, the robot creates a greedy up-left path and a greedy down-left path from t, within a window of height $4L$ centered at t. Note that the highest post in a $k \times M$ τ-fence-tree is $M\tau \leq 3L$ above the root (which is always distance L below t) and the lowest post is $k\tau \leq L$ below the root. So the robot is guaranteed to stay within a window of height $4L$ centered at t. Thus after the first trip, these greedy paths play the role of a wall; once the robot hits one of these paths, it can reach t with an additional cost that is only a low-order term in the total cost.

3.9 The Navigation Problem Viewed as a Hidden Task System

In this section we describe how a version of the robot navigation model of this chapter can be viewed as a hidden reversible task system (as defined in the previous chapter). Specifically we consider the "wall problem", i.e., the robot is required to make several trips between the point $s = 0$ and the line t defined by $x = n$ (n is an integer). For simplicity we assume that on each $t \rightarrow s$ trip the robot simply retraces its last $s \rightarrow t$ path. We also assume that all the (non-overlapping axis-parallel rectangular)

obstacles have width (i.e., the x-dimension) 1, and have corners at integer (x, y)-coordinates. Thus the robot can always move unhindered by obstacles vertically along any integer x-coordinate. Also, every point with integer (x, y)-coordinates can be reached by the robot.

This navigation model can be viewed as a reversible hidden task system (S, d). When the robot is at a point (i, j) we think of the online algorithm as being in state j after processing tasks $T_0, T_1, \ldots, T_{i-1}$, and task T_i is the next task to be processed. Thus the task T_i corresponds to the integer x-coordinate i, for $i = 0, 1, 2, \ldots, n$, and state j corresponds to integer y-coordinate j (and there are infinitely many of these). Changing the state from u to v corresponds to the robot moving vertically (i.e., without changing the x-coordinate) from $y = u$ to $y = v$. The state transition cost $d(u, v)$ is therefore defined in the natural manner as $|u - v|$. Processing task T_i in state j corresponds to moving from point (i, j) to $(i + 1, j)$ without changing the y-coordinate. This motion is impossible if there is an obstacle whose left side passes through (i, j) (and extends above and below (i, j)). If this is the case we define the cost of processing T_i in state j to be ∞, and otherwise we define this cost to be 1. In a similar manner, "undoing" task T_{i-1} in state j corresponds to moving from point (i, j) to $(i - 1, j)$ without changing the y-coordinate. If there is an obstacle whose right side passes through (i, j) we define the cost of undoing T_{i-1} in state j to be ∞, and 1 otherwise.

With these definitions it is clear that since the robot is tactile, $T_i(j)$ becomes known only when the algorithm is in state j and the next task is either T_i or T_{i+1} (in the latter case the robot finds out $T_i(j)$ by attempting to move backward toward s). Also it is easy to verify that the cost of processing task T_i in state j is the same as the cost of undoing task T_i in state j. The above navigation model is therefore a reversible hidden task system: each $s \rightarrow t$ trip of the robot can be viewed as processing the sequence of tasks $T_0, T_1, \ldots, T_n$.

For the k-trip problem, the performance metric $\rho(A, m, s, t)$ defined in Chapter 2 corresponds to the ratio $\rho(R, n, k)$ (s does not appear here since there are infinitely many states in S). Also in this case the ratio $\rho_i(A, m, s)$ corresponds to $\rho_i(R, n)$.

Thus in the language of task systems, we have shown the following results:

Theorem 11 *For a reversible hidden task system (S, d) where the state space consists of equally-spaced points on an infinite line (note that $s \equiv |S| = \infty$ in this case) and the task processing costs are all either 1 or ∞, the following results hold:*

1. *For any deterministic online algorithm A that processes a sequence of m tasks t times, $\rho(A, m, s, t) = \Omega(\sqrt{m/t})$.*
2. *There is a deterministic algorithm A for repeatedly processing a sequence of m tasks with the guarantee that for each $t \leq m$, $\rho(A, m, s, t) = O(\sqrt{m/t})$.*

3. *There is a deterministic algorithm A which for each $i \leq m$ achieves $\rho_i(A, m, s) = O(\sqrt{m/i})$.*

3.10 Conclusion and Open Problems

The core result of this chapter is an algorithm that performs a smooth tradeoff between search effort and the goodness of the path found. This algorithm may be of interest independently of the performance-improvement problem. For instance when a robot has more time or fuel available, one would like it to spend more effort and find a better path. The fence-tree structure is central to this search algorithm. Intuitively, one can think of the fence-tree as representing the collection of those obstacles in the scene which are responsible for making the scene difficult to cross from s to t. Thus the fence-tree in a sense captures the "essence" of a scene, as far as the difficulty (i.e., cost) of crossing the scene is concerned. It would be nice to explore whether an analogous structure can be defined in more general scenes. This might lead to a generalization of our results to such scenes.

At a higher level, our approach in designing a "learning" navigation algorithm was to start with an algorithm that achieves the above-mentioned cost/performance tradeoff, and convert that to a more incremental algorithm by spreading the work over several trips. This high-level idea may well be useful in designing performance-improvement algorithms for other tasks.

There are several other interesting research directions that can be explored. One question is whether the relatively complicated fence-tree-traversal strategy (especially for general axis-parallel rectangular obstacles) can be considerably simplified. What about extending our multi-trip results to more general scenes? Very recently, Berman and Karpinski [14] have designed a randomized $O(n^{3/4})$-competitive single-trip algorithm for 2-dimensional scenes containing arbitrary convex obstacles within which a unit circle can be inscribed. Achieving an $O(\sqrt{n})$ ratio for such scenes seems considerably harder. A good first step might be to consider scenes with rectangular obstacles in arbitrary orientations (i.e. not necessarily axis-parallel). It would also be interesting to extend Theorem 11 to more general task systems.

It is an open question whether a robot strategy that is allowed randomization can improve upon the $O(\sqrt{n})$ ratio for one trip. The best lower bound known is $\Omega(\log \log n)$ and was obtained by Karloff, Rabani and Ravid [47]. Independently of this question, it is conceivable that randomization could simplify our k-trip algorithm.

One can consider variations of the k-trip problem. One could examine how the robot can efficiently visit several destinations in a scene, improving performance wherever possible. Another possibility is to study how the robot can navigate (back and forth

between two points or otherwise) effectively in a scene that is changing slowly.

3.11 Related Work

Online Approaches to Complete Exploration

Our multi-trip algorithm improves its performance by gradually acquiring more information about the scene as it makes more trips. However, learning about the environment is only incidental to the performance-improvement objective. In fact our algorithm may not ever completely explore the scene. There has been considerable algorithmic work devoted to the problem of *completely exploring, or learning,* an unknown environment. We discuss below the work on learning geometric or graph environments. The next subsection focusses on learning environments that are naturally viewed as finite-state automata.

One approach researchers have taken to the problem of learning an unknown geometric environment or graph is to formulate it as an online problem: For a given class of environments, we must devise an algorithm that starts from a given point s and visits (or "sees") every point in the environment. The goal is to minimize the (worst-case) ratio of the length of this path to the optimal path length needed to verify a map of the environment if it were known. This is the familiar competitive ratio metric.

For instance Deng and Papadimitriou [30]consider the problem of exploring all edges of an unknown strongly-connected digraph. In their model the learner only knows how many unexplored edges emanate from the current node but does not know where they lead until it traverses them. There is a cost of 1 to traverse an edge. They show a lower bound and upper bound (i.e. an algorithm) of 2 on the performance ratio for Eulerian graphs. Their main result is an algorithm that achieves a ratio exponential in d (more precisely, $3(d+1)^{2d+1}$) where d is the number of edges that must be added to make the graph Eulerian.

In [29], Deng, Kameda and Papadimitriou examine the problem of designing a good strategy to walk in an unknown polygonal room with unknown opaque polygonal obstacles, such that every visible point in the room can be seen from some point on the walk. A peculiarity of this problem is that the offline problem in general is NP-hard. (Somewhat surprisingly, if the start and end points of the walk are the same, the problem, called the *watchman's route problem,* is polynomially solvable). They show a $2\sqrt{2}$-competitive (in the L_2 metric) algorithm for exploring the interior of a simple (not necessarily convex) rectilinear polygonal room that does not contain any obstacles. They show the same competitive ratio can be achieved for exploring the exterior of a rectilinear simple-polygonal room. They use these results to conclude that there is an $O(k)$-competitive algorithm for exploring the interior of a rectilinear

polygonal room containing at most k rectilinear polygonal obstacles. They are able to generalize this result to general polygons containing k general polygonal obstacles, with a very large constant factor (in the thousands) suppressed in the $O(k)$.

For the same problem, Kalyanasundaram and Pruhs [42] show a lower bound of $\Omega(\sqrt{n})$ on the competitive ratio, for the case of a scene containing n rectangular obstacles with a *bounded aspect ratio*. For these scenes they also show an algorithm that matches this lower bound.

Piecemeal Learning. Betke, Rivest and Singh [15] also focus on the goal of completely exploring an environment, but with a new restriction on the learner: The learning must be done *piecemeal* – that is, a bound B is given, and the learner is constrained to explore the environment in *phases;* each phase must start and end at the start point s, and have length at most B. This might model for instance a planetary exploration robot that must return to base every so often for refueling or to perform other tasks. The goal is to fully explore the environment while (nearly) minimizing the total distance walked. For the case of grid graphs with rectangular obstacles (equivalent to a rectangular room containing axis-parallel rectangular obstacles) they show an algorithm that piecemeal-learns the entire graph by walking a distance linear in the total length of the edges (i.e. the total obstacle-free area in the room). They assume that the bound B is at least 4 times the radius of the graph. They are able to generalize their results to arbitrary graphs [11].

Other approaches. Other authors have considered the map-learning problem ([68], [72]) but not from an on-line perspective. For instance in [68] the authors present an algorithm that incrementally builds a "visibility graph" of the environment as it is given pairs of points to travel between. Efficient exploration is not their main concern, however; they only show that their algorithm eventually converges to the correct visibility graph.

Inferring Automata

If it is known that an environment has a certain structure, this information could be exploited in order to learn it efficiently; it may be possible to *infer* the structure of unseen portions of the environment by experimenting with a portion of it. For instance, the unknown environment may have the behavior of a finite automaton, where the actions of the learner can change the state of the automaton. When viewed as state-transition diagrams, such environments can be seen to be a generalization of the graph environments described above. The goal of the learner would be to (efficiently) learn the structure of the automaton. Rivest and Schapire [74, 75] show an efficient algorithm for such a learner to infer the structure of any deterministic

finite-state automaton environment. The problem of inferring a finite automaton from its input-output behavior is a well-studied problem in learning theory [4, 35, 49, 50, 6, 8]. The work of Rivest and Schapire [74, 75] improves upon Angluin's [8] inference algorithm, which assumes that the learner has the ability to "reset" the automaton to some start state; the Rivest-Schapire algorithms do not require this assumption.

The problem of learning in an unknown environment has also been approached from a probabilistic viewpoint (see the survey [67]). The learner is modeled as a stochastic automaton operating in an unknown random environment, and the learning task is one of updating the action probabilities in accordance with inputs received from the environment.

Machine learning approaches to improving performance

The problem of improving performance with experience has received much attention in the machine learning community. One issue that researchers have focused on is the *exploration-exploitation tradeoff:* On the one hand the robot must explore unknown regions in order to find a better path, and on the other hand, it must exploit what it has learned so far in order to minimize its cost of learning. For instance P. C. Chen [24] considers how the computation *time* for path-planning in a known scene can be improved by making use of (portions of) solutions to previous path planning problems in the same scene. The author presents a learning algorithm and analyzes it in the PAC learning framework popular in computational learning theory. Problems similar to the multi-trip problem have been addressed in the framework of reinforcement learning. For example S. Thrun [80] presents heuristics for a robot to improve its path as it repeatedly travels between two points in a scene containing (possibly concave) obstacles. Only empirical results are provided, however. S. Koenig and R. Simmons [53] consider a similar problem on graphs, also in the reinforcement learning framework.

There has been work in robotics on heuristics for improving performance with experience. To mention only one such example, Christiansen [26] addressed the problem of how a robot can learn to predict the effects of its actions so that it can generate better plans for manipulation tasks.

Path-Planning amidst Unknown Obstacles

The work of Lumelsky and Stepanov [61, 60, 62] is very close in spirit to the navigation problems we consider. They consider the problem of navigating from a start point s to a point t in a scene filled with unknown *arbitrary* obstacles (possibly non-convex, non-polygonal). Because of the generality of the obstacles, however, only rather weak results can be shown in this case. They show that if the s-t distance is D, for any path-planning algorithm, and for any $\delta > 0$ (however small), there is a scene that can

force the algorithm to generate an s-t path of length at least $D + \sum_i p_i - \delta$, where the sum is over the perimeters of all obstacles that intersect a disc of radius D centered at t. They also show an algorithm (which they call "Bug1") that finds a path bounded by $D + 1.5 \sum_i p_i$. The authors do not consider the problem of improving performance by making multiple trips.

Arbitrary obstacles can form mazes. For examples of research on mazes, see Blum and Kozen [20] and Abelson and diSessa [1].

The Localization Problem

One aspect of the navigation problem we consider is that the robot always knows its current absolute coordinates, but does not have a (complete) map of the scene. Guibas, Motwani and Raghavan [36], and Kleinberg [52] and Akella [2] consider a variation that also lends itself naturally to an online formulation: Suppose that the robot *does* have a complete map of the scene, but does *not* know where it is in this scene, i.e. it does not know its coordinates relative to the scene. The robot is assumed to have vision. However, its snapshot of the world from a given position may be exactly the same if it were at any of a number of possible locations within the scene. To disambiguate its position, it must walk around and take snapshots until there is only one initial position consistent with these snapshots.

Guibas, Motwani and Raghavan [36] consider the problem of identifying all possible locations in a given scene that are consistent with a given snapshot. Kleinberg [52] considers the problem of devising good strategies to determine which of these possible locations is the true location – a task that he calls "localization". A good strategy is one which produces localizing paths whose ratio to the optimum localization path length is small. Kleinberg shows a lower bound of $\Omega(\sqrt{n})$ on the performance ratio of any algorithm (even randomized) for scenes containing n non-overlapping axis-parallel rectangular obstacles. For these scenes he also shows an algorithm that achieves a ratio of $O(n^{2/3})$. Computing a localizing strategy is similar to computing an adaptive distinguishing or homing sequence for a finite state machine [54].

Path-Planning with Complete Information

For completeness we include here a sampling of the previous research on robot path planning with complete information, also known as the *piano movers problem*. This problem is formulated as follows: Given a solid object in two or three dimensional space, of known size and shape, its initial and target position and orientation, and a set of obstacles of known shapes, positions and orientations in space, find a continuous obstacle-avoiding path for the object from the initial position to the final position (and orientation). Unlike the corresponding problem with unknown obstacles, the main

difficulty here is not in showing that a good path can be found, but in *efficiently* computing a feasible path. Reif [73] first showed that this problem is PSPACE-hard in general. Schwartz and Sharir [77] showed a polynomial-time algorithm that finds a path (or decides that a path does not exist) for the 2D case with convex polygonal obstacles. For other algorithmic research in this area see [59], [78],[39] and the references therein.

Deferred Data Structuring

Recall that our incremental multi-trip algorithm spreads the building of the fence-tree over several trips in order to achieve for each trip the optimal (up to constant factors) competitive ratio for that trip. This high-level idea resembles the approach used in certain query-processing algorithms. For instance Karp, Motwani and Raghavan [48] consider certain geometric search problems where a set of data items is given and a series of queries about the data set must be answered online (i.e., each query must be answered before the next query arrives). The usual approach to such search problems consists of preprocessing the data set to build a search structure that enables queries to be answered efficiently. However this approach is only economical if there is a sufficiently large number of queries. The authors present algorithms for gradually building the search structure "on-the-fly" as queries are answered, so that the performance is at least as good as the preprocessing approach, and is in fact better when the number of queries is small. They refer to this approach as *deferred data structuring.*

Chapter 4

Improving Performance in Paging

4.1 Introduction

Consider a two-level memory system consisting of a fast memory (the cache) of k pages and a slow memory of $n-k$ pages. Suppose there is a fixed set of n pages, each of which is either in the slow memory or the fast memory, and a program makes a sequence of requests for these pages. The requested page must be in the fast memory in order to be accessed. If the requested page is not in fast memory, there is a *page fault*, and the pager (typically in the kernel of the operating system) intercepts this request and brings in the requested page from slow memory to fast memory. If the cache is full, it must *evict* a page from fast memory in order to make room for the requested page.

In the *standard online paging problem* [32] the pager knows *all* the previous page requests (including those that do not result in page faults) in the sequence, and must decide which page to evict from fast memory, without *any* knowledge about future requests.

For a given paging strategy, its *competitive ratio* is defined as the worst case over all request sequences, of the ratio of the number of page faults suffered by the strategy to the number of page faults incurred by an optimum sequence of evictions on that request sequence. Sleator and Tarjan [79] have shown that two strategies for paging – evicting the least recently used page, or LRU, and first-in-first-out, or FIFO – have a competitive ratio of k, and that no deterministic online algorithm could achieve a factor less than k. Karlin, Manasse, Rudolph and Sleator [44] studied a different paging algorithm and also introduced the term "competitive".

Since the paging policy of an operating system cannot be equally good for all applications, some researchers [65, 25] have considered extending the operating system to allow the user to specfiy the page replacement policy. One issue in the design of

such systems is deciding whether or not to allow the user-level pager to monitor every single page request made by the program. For efficiency reasons and due to hardware limitations, it is common to only invoke the pager when there is a page fault [65]. Let us call such a restricted pager a *sleepy pager*; this is the model we are concerned with in this chapter. Figuratively speaking, the pager is "woken up" only when a page fault occurs, and after deciding which page to evict, it goes back to sleep and is thus unaware of what (or how many) requests are made before it is woken up again.

Let us note some aspects of the sleepy model. Needless to say, a sleepy pager is also *lazy* in the sense that it only throws out a page from the cache when there is a page fault. Since a sleepy pager is only aware of page faults, the FIFO strategy can be implemented on a sleepy pager, but the LRU strategy cannot. Another interesting fact about the sleepy model is the following. In the standard model, there is a randomized algorithm (the marking algorithm of [32]) that achieves a competitive ratio of $O(\log k)$, that is, the expected number of page faults on any request sequence is within an $O(\log k)$ factor of the optimum. However in the sleepy model even a randomized algorithm cannot achieve a ratio better than $k/2$. To see this, consider a paging scenario where $n = k + 1$, i.e., there is always exactly one page missing from the cache. Consider a request sequence consisting of phases, where each phase is generated as follows: pick a random i uniformly from $S = \{2, \ldots, k+1\}$, and generate a sequence of k subphases of requests where each subphase is the request sequence $\langle 1, 2, \ldots, k+1 \rangle$ modified by replacing request i by $i - 1$ (thus $i - 1$ occurs twice in a row). Clearly for any deterministic sleepy paging algorithm (even one that knows the number of page requests so far) in any given phase, the expected number of page faults is at least $k/2$, whereas the optimum number of page faults is just 1. Thus no deterministic algorithm can achieve a ratio better than $k/2$ on this randomized request sequence, and by the Min-Max principle [83] $k/2$ is a lower bound on the competitive ratio of any randomized algorithm.

We consider the following problem for a sleepy pager. Suppose that the program repeatedly generates the *same* sequence σ of page requests – this could easily happen for instance inside a loop of a program. On the first "iteration", since the pager does not know any of the page requests, the competitive ratio can be no better than $\Omega(k)$. However after the first iteration the pager has *partial information* about the request sequence, namely the sequence of page faults on the first iteration. The question is. can it exploit this information to improve its competitive ratio? Can it continue to improve, by acquiring more information with each iteration?

More formally, let us consider a sleepy pager that when woken up by a page fault, knows the "time" relative to the start of the request sequence, so that it knows where the page fault "fits in" relative to page faults it has seen previously. If the same finite request sequence σ is repeated several times, for a given sleepy pager strategy P, let $P^{(t)}(\sigma)$ denote the total number of page faults incurred by P during the first t

repetitions of σ, and let $OPT(\sigma)$ be the number of page faults incurred by an optimal offline strategy. The *average competitive ratio* of P at the end of t iterations is then defined as

$$\rho(P,n,k,t) = \sup_{OPT(\sigma)\neq 0} \frac{P^{(t)}(\sigma)}{t \cdot OPT(\sigma)}.$$

In other words $\rho(P,n,k,t)$ is the worst case over all (finite) σ (for which $OPT(\sigma)$ is not zero) of the ratio of the *average* page faults of P per iteration at the end of t iterations, to the optimal number of page faults in an iteration. Note that this definition is analogous to the one we made for the navigation problem.

In general one would like to design an online algorithm P for which $\rho(P,n,k,t)$ decreases rapidly with increasing t. As a first step toward this goal, in this chapter we consider the special case of the above problem where the total number of pages n is exactly $k+1$, that is, exactly one of the $k+1$ pages is always missing from the k-page cache. This is a well-studied special case [63]. For this case we simply write $\rho(P,k,t)$ for $\rho(P,n,k,t)$. This is essentially the case of k servers on $k+1$ points in a uniform space. For this case we design an algorithm P that has the following behavior: If the same finite request sequence is repeated some number of times, its average t-iteration competitive ratio $\rho(P,k,t)$ equals k for $t=1$, $O(\frac{k}{t}\log t)$ for each $t \leq k$, and $O(\log k)$ for $t > k$. Clearly from the previous remarks no algorithm (even a randomized one) can achieve an average competitive ratio better than $\Omega(k/t)$, so our results are within an $O(\log t)$ factor of the optimum.

The $n = k+1$ case of the paging problem corresponds to the k-server problem [63] on $k+1$ points, and can be viewed as the following game between the algorithm and an adversary, played on a uniform space of $k+1$ points (i.e. the distance between any two points is the same, say 1). We think of the algorithm as occupying the point corresponding to the page missing from the cache, and we think of the adversary as probing points of this space. We can think of the location of the algorithm as the "hole" since it is the one point not occupied by a server. When the adversary probes the point on which the algorithm stands, the algorithm must move to a different point, paying a cost of 1. We can think In the case of a sleepy pager, the algorithm is not aware of which point is probed by the adversary if it is different from the current location of the algorithm. This game is called the *pursuit-evasion game*, and the adversary is called the pursuer and the algorithm is called the evader. These terms were introduced by Blum, Karloff, Rabani and Saks in [18]. Figure 4.1 shows an example of a sleepy pager that moves the hole in a fixed cyclical order (equivalent to FIFO order) among all the pages.

For the sake of continuity we postpone the discussion of related work to Section 4.7.

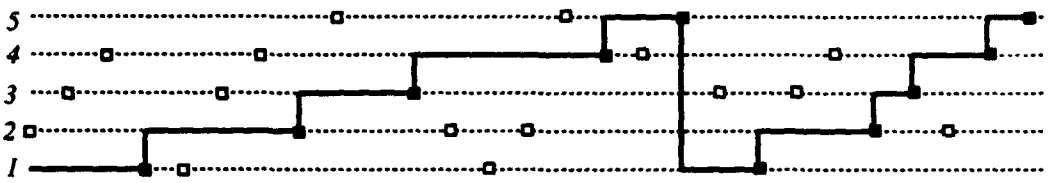

Figure 4.1: A sleepy pager that moves the hole in the cyclical order $\langle 1,2,3,4,5\rangle$. Each dotted horizontal line represents a page. The dark solid line represents the path of the hole (missing page). Each square is a page request; the darkened squares are requests that hit the hole (page faults), and are therefore the only ones the pager is aware of.

4.2 Preliminaries

At a given stage during any iteration of the (initially unknown) request sequence, some of the requests in the sequence will be known to the pager. In what follows, whenever we speak of requests we will mean the known requests. We will in fact abuse notation and use $\sigma = \sigma(1), \sigma(2), \ldots$ to stand for the current sequence of known requests at any time. Henceforth when we say "time" we will mean the time relative to the start of the actual request sequence that is being repeated. At any given stage, for $i < j$ we define a *cell* $[i,j)$ as the time period that starts just before request $\sigma(i)$ and ends just before request $\sigma(j)$. In particular the requests $\sigma(i), \sigma(i+1), \ldots, \sigma(j-1)$ belong to the cell $[i,j)$. If at a later stage a request is seen within this time period then that request is considered to belong to this cell. For convenience we write $[0,i)$ to denote a cell that starts at the beginning of the underlying actual request sequence and write $[i,\infty)$ to denote a cell that ends at the end of the underlying request sequence. Requests $\sigma(i)$ and $\sigma(j)$ are called the *boundaries* of the cell $[i,j)$. Note that a cell includes the left boundary but not the right boundary. Thus when two cells $[i,j)$ and $[j,h)$ are adjacent, the boundary request $\sigma(j)$ belongs to the right cell, and not to the left cell.

A *phase* is defined as a cell that contains $k+1$ distinct requests, i.e., all pages are requested at least once. Clearly an optimal offline pager must pay at least 1 (i.e. incur 1 page fault) in processing a phase of requests.

4.3 The Algorithm

At all times the algorithm uses a variable t to denote the number of iterations so far (including the current iteration). If $t > k$, our algorithm simply moves the hole in exactly the same way as it did in the best iteration (i.e. the one where the fewest faults were incurred) so far. Thus its competitive ratio on each iteration after k

iterations is no worse than the average competitive ratio $\rho(P,k,k)$ at the end of k iterations. We thus only need to describe our algorithm for $t \leq k$.

On the first iteration, the algorithm repeatedly moves the hole in a fixed cyclical order among the $k+1$ pages. Clearly then the page fault sequence at the end of this iteration can be partitioned into a sequence of phases (i.e. cells where every page is requested at least once).

On the second and later iterations, the algorithm uses the page fault sequence of the first iteration as a reference, in the following sense. The *span* of any cell is the number of first-iteration page-faults (i.e. requests that were discovered in the first iteration) that belong to that cell. For instance in the lower part of Fig. 4.2 the span of the cell whose boundaries are A and B is 2 (recall that the right boundary does not belong to the cell). A cell (or phase) is said to be *small* if it has span $s \leq k/t$, and it is said to be *big* otherwise.

At any stage, the request sequence consists of a sequence of cells, where the cells evolve in the following way. The algorithm responds to page faults differently depending on whether it is in a small cell or a big cell. In a small cell the algorithm merely moves the hole to wherever the hole was at this time during the first iteration. If the cell is big and is not a phase, the algorithm moves the hole to a page that is not known to be requested in this cell. As more requests (page faults) are discovered in a big cell, the cell may become a phase, i.e. every page is requested at least once. When a big cell becomes a phase $[i,j)$ the algorithm immediately splits it into two cells $[i,u)$ and $[u,j)$ such that the span of $[i,u)$ is $\lfloor s/2 \rfloor$ (and therefore the span of $[u,j)$ is $\lceil s/2 \rceil$). If a cell is small, as the number of iterations t increases, eventually its span may exceed k/t, in which case it is no longer small. Thus over a sequence of iterations the "evolution" of a cell may either look like:

$$\text{small cell} \longrightarrow \text{big cell} \longrightarrow \text{big phase} \longrightarrow \text{split into 2 cells,}$$

or like

$$\text{big cell} \longrightarrow \text{big phase} \longrightarrow \text{split into 2 cells.}$$

Therefore at any stage the sequence of known requests consists of a sequence of (small or big) cells.

We are now ready to describe our algorithm. The first three iterations of this algorithm are illustrated in Fig. 4.2.

Algorithm SleepyPager:

At any stage: Let $t \leq k$ denote the number of iterations so far, including the current one. Whenever a big cell (i.e. of span $> k/t$) becomes a phase (i.e. every page is requested at least once in the cell), divide it into two roughly equal-span cells as above. Thus at any time the (known) request sequence consists of a sequence of (small or big) cells.

First Iteration: Fix an order on the pages, and move the hole cyclically in this order. Divide the request sequence σ into cells $[0, a_1), [a_1, a_2), [a_2, a_3), \ldots, [a_m, \infty)$ so that the span of each cell is $k+1$, except possibly for the last cell.

Each remaining iteration: When the hole is hit by a request,

(a) If it is in a big cell then move the hole to a page not known to be requested in this cell;

(b) If it is in a small cell then move the hole to wherever the hole was at this time during the first iteration (if it is not already there).

4.4 Analysis

For any $t \leq k$, let us consider the behavior of the algorithm during the first t iterations of the request sequence. The algorithm induces a natural rooted forest structure on the cells formed at various stages. The root of each tree (which we call a *cell tree*) in the forest represents a phase at the end of the first iteration, and when a cell splits into two cells, the corresponding node has two children representing those cells. Thus a cell tree is a full binary tree, i.e., a node either has no children or two children. Clearly the leaves of the forest stand for the cells (big or small) at the end of the t'th iteration. An example of a cell tree is shown in Fig. 4.3.

A cell-tree has the property that cells corresponding to two nodes overlap exactly when one is an ancestor of the other. The internal nodes correspond to the cells that became phases at some point during the algorithm. We would like to bound the number of nodes in a cell tree in terms of the number of disjoint phases. To this end, let us define a *petal* in a cell tree as a node that has two children, both of which are leaves (See Fig. 4.3). Since none of the petals are ancestors of each other, the number of petals is a lower bound on OPT, the cost of an optimal offline pager on the underlying request sequence. Also note that since a petal cell eventually turns into a big phase and splits into two cells, its span must be greater than k/t.

Lemma 12 *Assume $t \leq k$. If p is the number of petals in a cell tree at the end of t iterations. Then there are at most $(4p + 2p\lceil \lg t \rceil + 1)$ nodes in the tree.*

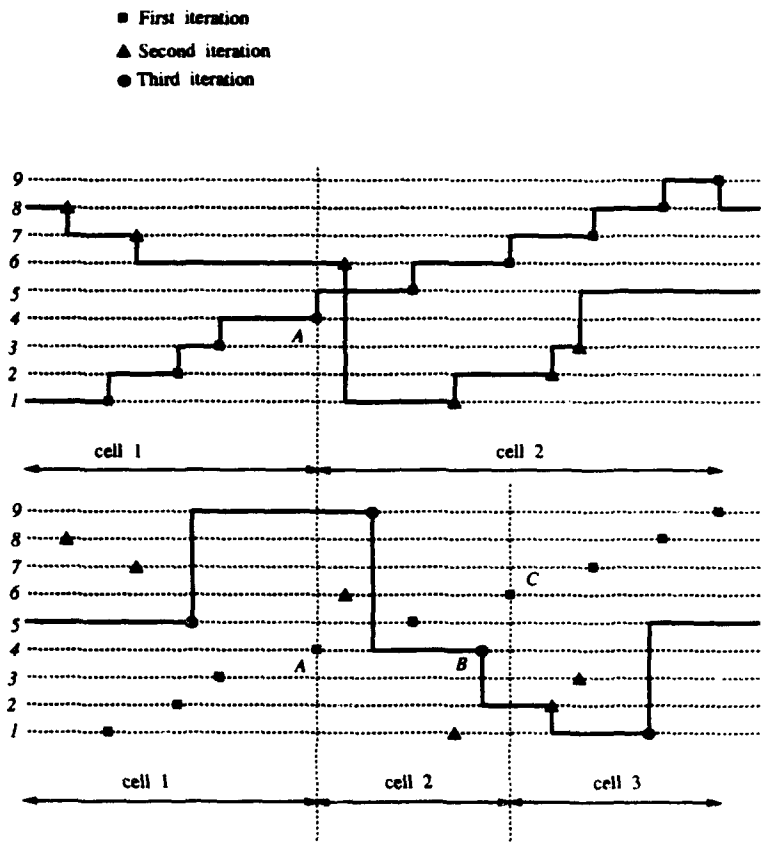

Figure 4.2: The first three iterations of algorithm **SleepyPager**, with 9 pages (i.e. $k = 8$). Only the requests (i.e. page faults) seen by the pager are shown. The top figure shows the hole-paths of the first two iterations. After the first iteration, there is just one phase, and this phase is divided into two cells (of spans 4 and 5) with common boundary A. After the second iteration, neither of the cells becomes a phase. On the third iteration (bottom figure), when the hole is hit by request B, cell 2 becomes a phase, and is immediately split into two cells (now called cells 2 and 3) at request C.

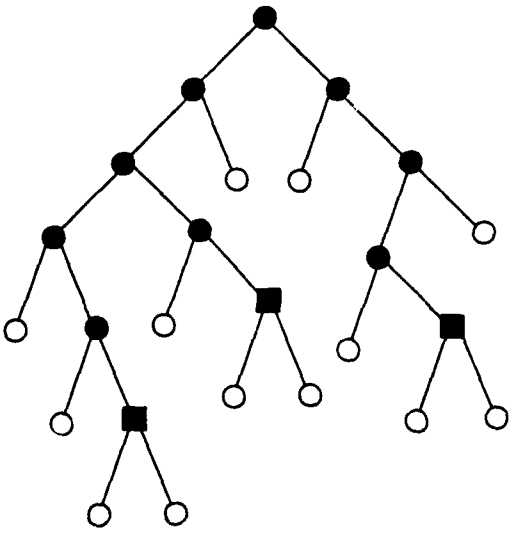

Figure 4.3: A cell tree. Square nodes are petals.

Proof: Since the number of leaves in a cell tree equals 1 plus the number of internal nodes, it suffices to bound the number of internal nodes. Each internal node is either a petal or an ancestor of a petal. We claim that for any petal, the number of nodes on the path from the root to the petal (including both) is at most $2 + \lceil \lg t \rceil$. This is because the root node has span at most $k+1$, a petal has span greater than k/t and each splitting of a cell of span s produces cells of span at most $\lceil s/2 \rceil$. This implies there are at most $p(2+\lceil \lg t \rceil)$ internal nodes in a cell tree. The total number of nodes, including the leaves, is therefore at most $2p(2+\lceil \lg t \rceil)+1$, which implies the lemma. ■

We are now ready to show

Theorem 13 *Algorithm* SleepyPager *has the following behavior: For every t, the average t-iteration competitive ratio $\rho(P,k,t)$ is at most $42\frac{k}{t}\lceil \lg t \rceil$ when $t \leq k$, and at most $42\lceil \lg k \rceil$ when $t > k$.*

Proof: Let $t \leq k$. As in the above lemma let p be the number of petals in the cell-forest associated with this algorithm. Then by that lemma there are at most $4p + 2p\lceil \lg t \rceil + 1 \leq 7p\lceil \lg t \rceil$ nodes in the cell-forest (assuming $k, t \geq 2$).

Suppose σ is the request sequence that is being repeated t times. Clearly $OPT(\sigma) \geq p$.

We wish to bound the total number $P^{(t)}(\sigma)$ of all requests that hit the hole path during the first t iterations. At any time the hole may be hit either by a previously-seen request (call this an old-hit) or by a new request (call this a new-hit). Moreover these hits can occur in either a big cell or a small cell. Note that when the hole *first*

enters a cell, it may be hit by a request, either old or new. Let us call such a hit an initial-hit. Even if there is an initial-hit in every cell on all t iterations, the total number of these is at most $7pt\lceil \lg t \rceil$.

After the initial hit, (a) if the hole is in a big cell, it moves to a page *not* requested in the current cell, and (b) if the hole is in a small cell, it moves to the first-iteration hole path and follows it. There can be at most $k+1$ non-initial new-hits in a big cell before it turns into a big phase and splits, so the total non-initial new-hits in all cells over all t iterations is at most $(k+1)$ times the number of nodes in the ceil forest, which is at most $(k+1)\cdot 7p\lceil \lg t \rceil$. The old-hit cost of following the first-iteration path in a small cell is just the span of the critical phase. Notice that a small cell of span k/x cannot stay small for more than $\lceil x \rceil$ iterations, so the total of this cost in a given small cell, over all iterations, is at most $(k/x)(x+1) \leq 2k$, independent of x. This adds up to at most $2k \cdot 7p\lceil \lg t \rceil$ over all small cells.

Therefore the total cost $P^{(t)}(\sigma)$ is at most

$$(t+k+1+2k).7p\lceil \lg t \rceil \leq 21(k/t+1)\lceil \lg t \rceil \cdot OPT,$$

which implies $\rho(P,k,t) \leq 21(k/t+1)\lceil \lg t \rceil \leq 42\frac{k}{t}\lceil \lg t \rceil$.

On each iteration after the first k iterations, the competitive ratio of the algorithm is at most the average competitive ratio at the end of k iterations, so clearly for $t > k$ the algorithm achieves $\rho(P,k,t) \leq 42\lceil \lg k \rceil$. ■

4.5 The Paging Problem Viewed as a Hidden Task System

We already pointed out in Chapter 4 how the standard paging problem (with $n = k+1$) can be viewed as a task system (S,d) with a uniform state space S. Recall that page request i corresponds to a "task" i, and state j represents the cache configuration where page j is the (only) page missing from the cache. Thus the cost of "processing" task i in state j is 0 if $i \neq j$ and it is ∞ otherwise. Such tasks where the processing cost is ∞ in one state and 0 in all others are called ∞-*elementary*, a term that was introduced by Borodin, Linial and Saks [23].

The sleepy pager model (with $n = k+1$) can be viewed as a hidden task system on a uniform state space. Indeed, if at any stage the next "task" is j, unless the sleepy pager is in the state j it does not know in which state this task costs ∞ to process. (Of course, if it is in state j it knows that the next task costs 0 to process in every other state). Note that unlike in a general reversible hidden task system a sleepy pager does not even have the option of probing several states to find the ∞-cost state.

The performance metric $\rho(A, m, s, t)$ for a hidden task system thus corresponds to the ratio $\rho(P, k, t)$ for a sleepy pager. In other words we have shown the following result:

Theorem 14 *For a hidden task system (S, d) with $s = |S|$ where the state space is uniform and the possible tasks are ∞-elementary, there is an online algorithm A for repeatedly processing a sequence of m tasks, with the guarantee that $\rho(A, m, s, t) = O(\frac{s}{t} \log t)$ as long as the number of repetitions t is no more than s and $\rho(A, m, s, t) = O(\log s)$ for $t > s$. (Our performance guarantees turn out to be independent of m.)*

4.6 Conclusion and Open Problems

An obvious open problem is whether a performance-improvement algorithm can be designed for the general paging problem where n could be (much) bigger than $k + 1$. Even for the special case $n = k + 1$, there is a gap between the upper and lower bounds on $\rho(P, k, t)$, and it would be interesting to improve either. Note that we only know of the trivial lower bound $\Omega(k/t)$.

We have designed our algorithm for situations where the *same* request sequence is repeated. An intriguing question is whether one can design algorithms that improve their competitiveness even when the request sequence is *almost* the same.

In the framework of task systems, one can consider generalizations of the sleepy pager problem. For instance we can extend Theorem 14 to the case of arbitrary task processing costs (but the state space is still uniform). Whether a similar result can be shown for non-uniform state spaces is an open question.

4.7 Related Work

Our sleepy paging algorithm uses partial information from prior executions to improve its (average) competitiveness. Other researchers have studied how an online algorithm can exploit information about its future input, although their work differs from ours in both the type of information and the manner in which it is acquired.

The Influence of Lookahead

One type of partial information researchers have analyzed is *lookahead* information. In the context of paging, Albers [3] and Young [84] consider how the competitive ratio can be improved when the online algorithm sees not only the current page request but also some future requests. Such information may be available in many real computer

systems, for various reasons [3]: the page requests may arrive in blocks; or if several processes are running, some of them may accumulate several page faults and wait for service; or a memory system with prefetching might demand not only the currently accessed page but other possibly related pages to be in fast memory.

Specifically, Albers makes a distinction between *weak lookahead* and *strong lookahead*. An algorithm is said to have weak lookahead of size $l \geq 1$ if it sees the present request and the next l requests. It is well known that this type of lookahead cannot help improve competitiveness since an adversary can render lookahead useless by replicating each request l times. An algorithm has strong lookahead of size l if it sees the next s requests, where s is the smallest number such that the cardinality of the set of the next s requests is $l+1$. Under this model, Albers [3] presents a variant of LRU that, given a strong lookahead of size l, where $l \leq k-2$, achieves a competitive ratio of $(k-l)$. She also shows that no deterministic algorithm can do better. Albers presents a randomized algorithm (a modification of the Marking algorithm of [32]) which when given strong lookahead with $l \leq k-2$, is $2H(k-l)$-competitive, where $H(k) = \sum_{i=1}^{k} 1/i$ denotes the kth harmonic number. Further, she shows that $H(k-l)$ is a lower bound on the competitive ratio of such a paging algorithm.

Young [84] studies online paging algorithms that have *resource-bounded lookahead of size l*. Here the pager sees the present request and the maximal sequence of future requests for which it will incur l faults. He presents deterministic and randomized algorithms with this type of lookahead which are $\max\{2k/l, 2\}$-competitive and $2(\ln(k/l)+1)$-competitive respectively.

Researchers have also studied the influence of lookahead in online algorithms for certain dynamic location and online graph problems [40, 27, 43, 37].

Exploiting Information about Request Patterns

Recently some authors have addressed the question of whether a paging algorithm can improve its competitiveness by exploiting knowledge about a program's memory access patterns. For instance programs are known to exhibit *locality of reference*, and analyzing paging algorithms without taking this account might result in predictions which do not match with practice. The LRU paging algorithm is a case in point: it has been observed in practice to perform much better than one would expect from the theoretical lower bound of k on its competitive ratio (k is the number of cache pages).

The work of Borodin, Irani, Raghavan and Schieber [21] and Irani, Karlin and Phillips [41] is an attempt to address this mismatch between theory and practice. They model the program's locality of reference by an *access graph* G that is known to the algorithm. The nodes of G (which may be either directed or undirected) represent

pages that a program can reference. The edges of G constrain the possible request sequences that can be generated by the program, in the sense that a request to page v can follow a request to page u only if either $u = v$ or (u, v) is an edge of G. The definition of competitiveness remains the same, except that only request sequences consistent with G are considered. They show several results, including tight bounds on the performance of LRU on any access graph G, and that LRU is optimal among online algorithms for the important special case when G is a tree. They also show that for small k, LRU is close to optimal on any G.

Adaptive Prefetching

In applications such as databases and hypertext systems, there is sufficient time between page requests to *prefetch* pages into the cache even before they are requested. Since prefetching can be done "for free" (i.e. with no visible delay to the user), the only cost of a prefetching algorithm is the number of page faults. A good prefetching algorithm must thus be able to accurately *predict* the next few page requests. A prefetching algorithm cannot be expected to be competitive in the usual sense since an offline algorithm would always prefetch the next page in the request sequence, and would *never* suffer page faults. In [82], Vitter and Krishnan introduce a model where the page requests are generated by a Markov source, and a prefetching algorithm is evaluated relative to the best *online* algorithm that has complete knowledge of the underlying Markov source. They are able to design prefetching algorithms whose expected fault rate converges to the minimum expected fault rate as the request sequence length approaches infinity. Their algorithms are based on an innovative use of the classical Ziv-Lempel [85] data compression technique, the intuition being that good data compression requires good prediction.

Others have taken an empirical approach to prefetching. For instance, Palmer and Zdonik [69] use a neural network approach to prediction, and Salem [76] uses prediction based on various first-order statistics.

Other Work on Improving Paging Performance

McFarling [64] has considered the possibility that the instruction-cache hit rate may be improved by deciding smartly whether or not to bring in an executed instruction into a full cache. He studied some ways of making this decision using a small finite state machine to recognize the common instruction reference patterns.

Kilburn, Edwards, Lanigan and Sumner [51] empirically studied some heuristics to improve paging performance in the Atlas computer. Their scheme, which they refer to as a "learning" program, decides which page to evict from the cache based on the reference frequencies of the cache pages. They speculate on the possibility that a

paging program could attempt to predict which pages might be required by the application program in the near future, and improve its prediction accuracy by recording its successful and unsuccessful predictions.

Chapter 5

Online Learning of Switching Concepts

5.1 Introduction

Consider the following scenario. Alice and Bob like seeing movies. Alice, being the thoughtful type, one day decides to come up with a simple rule that predicts whether or not Bob will like a particular movie. An example of such a rule might be: "IF the movie is black-and-white, OR Clint Eastwood stars in it, THEN Bob likes it". Alice knows that if there is a relatively simple rule that models Bob's tastes at all times, she can use standard algorithms to infer (a good approximation to) that rule. To her dismay however, Alice finds that no single, simple rule seems to consistently explain Bob's movie tastes at all times.

It occurs to Alice that perhaps Bob's movie tastes change depending on his mood. Let us suppose in fact that there are *two* simple rules that predict Bob's reaction to a movie: One applies when he is in a good mood, the other when he is in a bad mood, and his mood changes every now and then. However, Bob likes to be mysterious so he never reveals his mood to Alice. So Alice is confronted with the following problem: based on observations about Bob's movie preferences, she must infer the two simple rules that explain Bob's tastes.

The standard problem considered in machine learning theory is that of learning a fixed concept from some class C. The problem confronting Alice is a generalization of this standard problem, where the concept may switch between a small number (two, in this case) of different concepts during the course of learning. We introduce our switching concepts model in the next section. In order be able to get to this right away, we postpone the discussion of related work to the last section of this chapter.

5.2 Notation and Model

Let V be a set of n variables $\{x_1, \ldots, x_n\}$. An example x is an assignment of 0 or 1 to each $x_i \in V$, and let X be the set of all examples. We will often just write $x_i = 1$ or $x_i = 0$ to mean $x(x_i) = 1$ or $x(x_i) = 0$ when the example is clear from context. A labeled example is a pair (x, l) where $x \in X$ and $l \in \{0, 1\}$. A concept is a boolean function over examples and a concept class is a collection of concepts. For a disjunction c, define $R(c)$ to be the set of all variables disjoined in c. So, $R(c)$ is the set of *relevant* variables.

In our model of switching concepts, for a given concept class C (such as monotone disjunctions) we consider the following game played by Alice (the learner) and Bob (the adversary). At the start of the game, Bob selects two secret concepts c_1, c_2 from C. (In our example above, X is the class of movies, c_1 and c_2 are the two rules that model Bob's movie preferences depending on his mood). In each round of the game, first Bob picks $\mathbf{c} = c_1$ or $\mathbf{c} = c_2$ as his current "active" concept, and he does not reveal to Alice which concept he picks (i.e., Bob picks his mood).

Then Alice either:

1. (**Query**). Chooses an $x \in X$ and asks Bob for the value of $\mathbf{c}(x)$ (which Bob answers correctly), or
2. (**Prediction**). Asks Bob to choose an $x \in X$ and then guesses the value of $\mathbf{c}(x)$.

Each query (in step 1) or prediction mistake (in step 2) costs Alice 1 unit. Alice's goal is to minimize her cost. Note that in this model, the adversary may decide to switch based on the interaction so far, but cannot switch, for instance, between receiving a membership query and responding to it.

Informally, Alice makes the queries in order to glean information about Bob's secret concepts c_1 and c_2, so that eventually she can make more correct predictions.

If $c_1 = c_2$ then this game reduces to the standard "mistake bound with membership queries" or "(extended) equivalence and membership queries" models: for various concept classes C, there are algorithms that Alice can use whose cost in any game is bounded by a polynomial (in n).

However for $c_1 \neq c_2$, it is not hard to see that for any deterministic strategy that Alice uses, Bob can force Alice to make a mistake in every prediction round: Bob picks an example x such that $c_1(x) \neq c_2(x)$, and in each prediction round, if Alice's prediction would agree with the current concept c, then Bob switches to the other concept. In fact we show below that there is not even a randomized strategy whose expected cost stays within a factor strictly less than 1 of the number of switches so far (even up to an additive term)

Lemma 15 *Suppose Alice knows c_1 and c_2, and $c_1 \neq c_2$. Then for any given randomized strategy of Alice, there is a strategy of Bob such that for any constants (i.e. independent of the length of the game) $0 < \delta < 1$ and a, after sufficiently many rounds of the game, the expected number of mistakes plus queries of Alice exceeds $a + (1-\delta)s$, where s is the number of switches Bob has made so far.*

Proof: We can assume that Bob knows Alice's randomized strategy. Since $c_1 \neq c_2$, there is a y such that $c_1(y) \neq c_2(y)$. In a prediction round, Bob always gives the example y to Alice. In the ith round of the game, let the current concept (c_1 or c_2) be denoted by c, and let p_i be the probability that Alice makes a prediction and it equals $c(y)$, and q_i be the probability that Alice makes a query (So the probability that Alice makes a prediction and it does not equal $c(y)$ is $1 - p_i - q_i$). Bob knows p_i and q_i. Thus the probability that Alice makes a mistake in round i is p_i if Bob makes a switch at the start of that round, and it is $1 - p_i - q_i$ otherwise. In other words the expected cost (mistake plus query) ϕ_i of Alice in round i is $p_i + q_i$ if Bob makes a switch, and $1 - p_i$ otherwise.

For any j let Φ_j denote $\sum_{i=1}^{j} \phi_i$, and let s_j denote the number of switches Bob has made in the first j rounds. Note that Φ_j is the expected cost of Alice after j rounds of the game.

In each round Bob must decide whether to switch or not. The goal of Bob's strategy is to always maintain the invariant that $\Phi_i \geq s_i$. Just before the ith round starts, Bob knows Φ_{i-1} and s_{i-1}. If Bob switches, then $\phi_i = p_i + q_i$, and so $\Phi_i = \Phi_{i-1} + p_i + q_i$ and $s_i = s_{i-1} + 1$. Bob's strategy is to switch if and only if

$$\Phi_{i-1} + p_i + q_i \geq s_{i-1} + 1,$$

and this clearly maintains the invariant.

We claim that when the game is played with the above strategy of Bob, after some finite number of rounds u, the number of switches s_u exceeds a/δ. At this point clearly $s_u > a + (1-\delta)s_u$, and by the above invariant $\Phi_u > a + (1-\delta)s_u$, which is what we wanted to show.

To show that s_i exceeds a/δ for some finite i it suffices to show that there can never be more than a finite sequence of rounds where Bob does not switch. Consider some round, say round i, where Bob does not switch. This must mean that $\Phi_{i-1} + p_i + q_i < s_{i-1} + 1$, which implies $1 - p_i - q_i > \Phi_{i-1} - s_{i-1}$. The expected cost of Alice in round i is $1 - p_i$, so $\Phi_i = \Phi_{i-1} + 1 - p_i > \Phi_{i-1} + \Phi_{i-1} - s_{i-1}$. Since $s_i = s_{i-1}$, it follows that $\Phi_i - s_i > 2(\Phi_{i-1} - s_{i-1})$. Thus if there is no switch in round i, the difference $\Phi_i - s_i$ is strictly greater than twice the corresponding difference before this round. Thus after a finite number of rounds without switches, $\Phi_i - s_i$ must be at least 1, and Bob's strategy would dictate that he must switch. The claim then follows. ■

Thus there is no strategy for Alice with a bounded expected cost. What then should be a criterion for a good strategy? We adopt the popular *competitiveness* style of analysis from online algorithms. That is, we compare the performance of Alice's strategy with that of an "optimal" algorithm that knows both c_1 and c_2. Indeed, if Alice knows both c_1 and c_2, then she can use the following simple strategy to pay at most 1 whenever Bob switches his active concept: predict according to c_1 until a mistake, then switch to c_2, and so on. We call this the "standard steady-state algorithm". From the previous remarks, such a strategy is optimal. Thus we say that Alice's strategy is *t-competitive* if for any given $0 < \delta < 1$, in any game where Bob makes s switches, with probability at least $(1 - \delta)$ the cost of the strategy is bounded by $p(n, 1/\delta) + ts$ where p is a polynomial. One would like to design t-competitive algorithms for small constant t. This model can be easily generalized to switching among k different concepts from the class $\mathcal{C}$.

In the next section we present a 1-competitive randomized algorithm for learning a pair of monotone disjunctions in this model. Specifically, given $\delta > 0$, with probability at least $1-\delta$ the number of mistakes plus queries of the algorithm is $O(n^2 \log(n/\delta))+s$. The algorithm's running time per round is polynomial in n and $\log(1/\delta)$.

5.3 The Algorithm and Analysis

We begin with a high-level description of the algorithm. In its first stage, the algorithm makes a large number of queries, selected randomly according to a specific distribution. With high probability it learns one concept exactly and learns the other exactly as well if it has been used for classification a fair fraction of the time. Of course, it is possible that one concept has been used only very infrequently for classification, so one cannot hope to always learn both.

In the second stage, the algorithm predicts according to the first concept c_1 learned, and whenever its prediction is wrong, makes a query, hoping to gain information about the second concept c_2. We show that with high probability, after making sufficiently (polynomially) many queries to c_2, the algorithm will both recognize this fact—so it can stop making the queries—and exactly learn c_2. We also show that except for mistakes and queries associated with the queries to c_2, the algorithm makes at most one mistake and one query for every *two* switches of the adversary, which makes it 1-competitive.

In the analysis of our algorithm we use the following Chernoff-type bounds from [9]. When there are m independent trials of a Bernoulli random variable with probability of success p, then for any $0 < \beta < 1$,

$$\mathbf{Pr}[\text{there are fewer than } (1-\beta)mp \text{ successes}] \leq e^{-\beta^2 mp/2}, \text{and}$$

$$\mathbf{Pr}[\text{there are more than } (1+\beta)mp \text{ successes}] \leq e^{-\beta^2 mp/3}.$$

For convenience, let $\langle x_i \rangle$ denote the example that sets only x_i to 1, and similarly define $\langle x_i, x_j \rangle$ as the example that sets only x_i and x_j to 1. For a monotone disjunction c_i we write $R(c_i)$ to denote the set of variables relevant to c_i, i.e., the set of variables such that $c_i(x)$ is 0 if and only if all the variables in $R(c_i)$ are set to 0 in x. The first stage of the algorithm is as follows.

Algorithm Query-Learn, Stage One:

Given: $\delta > 0$.

1. *Make* $m = kn^2 \log(n/\delta)$ *(for* $k = 5400$*) queries each selected independently according to the following distribution:*

   Each example $\langle x_i \rangle$ *is selected with probability* $\frac{1}{2n}$. *Each example* $\langle x_i, x_j \rangle$ $(i \neq j)$ *is selected with probability* $\frac{1}{n(n-1)}$.

2. *For each* i *calculate* $f^+(x_i) =$ *(number of queries* $\langle x_i \rangle$ *that were classified as positive* $)/(\frac{m}{2n})$.

3. *If no index* i *has* $f^+(x_i) \in [\frac{1}{4}, \frac{3}{4}]$, *then let* h *be the disjunction of all* x_i *such that* $f^+(x_i) > 1/2$. *Output* h.

4. *Otherwise, pick* i *such that* $f^+(x_i) \in [\frac{1}{4}, \frac{3}{4}]$.

   Let h *be the disjunction of all* x_j *such that query* $\langle x_i, x_j \rangle$ *was never classified as negative.*

   Let h' *be the disjunction of all* x_j *such that either all queries* $\langle x_j \rangle$ *were classified as positive, or else* $f^+(x_j) > 0$ *and* $x_i \notin R(h)$.

   Output both h *and* h'.

Lemma 16 *For any adversary strategy, for any* $\delta > 0$, *if the two target concepts are monotone disjunctions then with probability* $1 - \delta$ *Algorithm* Query-Learn, Stage One *will output either one or both of the target concepts*

Proof: Algorithm Query-Learn, Stage One makes m queries. Since each query is selected independently from a fixed distribution, and the algorithm ignores the order of the examples, the only relevant choice of the adversary is the number m_1 of queries for which c_1 is used for classification. This can be seen by thinking of the algorithm as having two random tapes, one used when the target is c_1 and the other when the target is c_2. Let us first suppose that the adversary chooses m_1 *in advance*, and without loss of generality let us say $m_1/m \geq 1/2$.

For each $x_i \in R(c_1)$, the expected value of $f^+(x_i)$ is at least $m_1/m \geq 1/2$. (The expected value is 1 if $x_i \in R(c_2)$ as well.) Similarly, for each $x_i \notin R(c_1)$, the expected value of $f^+(x_i)$ is at most $1 - m_1/m \leq 1/2$. Chernoff bounds imply that with probability $1 - ne^{-\frac{1}{64}(m/n^2)} > 1 - \delta/5$, no $f^+(x_i)$ for $x_i \in R(c_1)$ will be less than $1/4$, and no $f^+(x_i)$ for $x_i \notin R(c_1)$ will be greater than $3/4$. Thus, with high probability, if the algorithm exits in Step 3 then $h = c_1$. Notice also that if $m_1 > 7m/8$, then with high probability (another $1 - \delta/5$) each $x_i \in R(c_1)$ has $f^+(x_i) > 3/4$, and all others have $f^+(x_i) < 1/4$, so the algorithm *will* exit in Step 3. Thus, we only need to analyze Step 4 for $m_1 \leq 7m/8$.

If the algorithm continues to Step 4, then with probability at least $(1 - \delta/5)$ the variable x_i chosen such that $f^+(x_i) \in [1/4, 3/4]$ is relevant to exactly one of the two concepts (if it was relevant to both, then with high probability $f^+(x_i)$ is very near 1); let us denote this concept by c and the other concept by c'. Notice that at least $m/8$ queries are classified by c'. In that case, for each $x_j \in R(c')$, the query $\langle x_i, x_j \rangle$ will never be answered negative. So, hypothesis h will contain all variables in $R(c')$. In addition, if $x_j \notin R(c')$, then the query $\langle x_i, x_j \rangle$ *will* be answered negative if it is asked when classification is according to c'. Since at least $m/8$ queries are classified according to c', this occurs with probability at least $1 - [1 - \frac{1}{n(n-1)}]^{m/8}$ which is at least $1 - \delta/5n$ for our choice of the constant k. So, with high probability $h = c'$. Given that this occurs, with high probability (at least $1 - \delta/5$) $h' = c$.

So far we have been assuming that m_1 was chosen in advance, but in fact for our choice of the constant $k = 5400$, each of the above failure rates is sufficiently small (less than $\delta/5m$) that for all values of $m_1 \leq m$ simultaneously, the chance of failure is at most δ. That is, if we think of the algorithm as having two random tapes as described above, then even if the adversary determines m_1 after seeing the entire contents of the tapes, it still can only foil the procedure with probability δ. ■

The second stage of the algorithm is as follows.

Algorithm Query-Learn, Stage Two:

Given: concept $c_1, \delta > 0$. (If we are given both concepts, we just run the standard steady-state algorithm).

1. $S \leftarrow 0$
2. *Repeat until* $S = n \log(2n/\delta)$:
   (a) *Predict according to c_1 until a mistake is made.*
   (b) *Make a query $\langle x_i \rangle$, with x_i chosen uniformly at random from $\{x_1, x_2, \ldots x_n\}$.*
   (c) *If the answer to the query is not the same as $c_1(\langle x_i \rangle)$, then $S \leftarrow S + 1$.*

3. *Let h_2 be the disjunction of the following variables:*
   - *Variables $x_i \notin R(c_1)$ such that query $\langle x_i \rangle$ was classified as positive at least once, and*
   - *Variables $x_i \in R(c_1)$ such that query $\langle x_i \rangle$ was* **always** *classified as positive.*
4. *Run the standard steady-state algorithm on c_1 and h_2.*

Lemma 17 *For any adversary strategy and pair (c_1, c_2) of monotone disjunctions such that c_1 is given to the learner, with probability $1 - \delta$ Algorithm* Query-Learn, Stage Two *will make at most $8n^2 \log(2n/\delta)) + s$ mistakes plus queries where s is the number of switches of the adversary.*

Proof: If $c_1 = c_2$, the algorithm stays in step 2(a) and the claim trivially holds. So, assume $c_1 \neq c_2$, so that there is at least one variable in the symmetric difference $R(c_1)\Delta R(c_2)$. For $i = 1, 2$, we say that a query is a c_i-query when the adversary uses concept c_i to classify it. Also, a c_2-query x is "informative" when the answer is different from $c_1(x)$. The counter S in step 2 counts precisely the number of informative c_2-queries made. Since $|R(c_1)\Delta R(c_2)| \geq 1$, if a c_2-query is made in step 2(b), there is at least a $1/n$ chance that it is informative. So, if we consider a sequence of $8n^2 \log(2n/\delta)$ c_2-queries selected according to the distribution of step 2(b), with probability at least $(1 - \delta)$ both (A) at least $n \log(2n/\delta)$ are informative, and (B) within the first $n \log(2n/\delta)$ c_2-queries, informative queries $\langle x_i \rangle$ have been made at least once for each $x_i \in R(c_1)\Delta R(c_2)$. This implies that with probability at least $(1 - \delta)$ either the algorithm proceeds to step 3 and by (B) the hypothesis h_2 created in step 3 exactly equals c_2, or else the adversary does not allow the algorithm to go to step 3 by not letting it make $8n^2 \log(2n/\delta)$ c_2-queries.[1]

Thus, it remains only to show that the number of mistakes plus queries made in step 2 is bounded by the number of adversary switches plus a constant times the number of c_2-queries. To see this, consider a sequence $\langle m_1, q_1, m_2, q_2, ... \rangle$ of mistakes and queries. Each mistake m_i is made while the classification is according to c_2. So, if q_i is a c_1-query then the adversary must have switched to c_1 for q_i and then back to c_2 before m_{i+1}, and we can charge both q_i and m_{i+1} to the switches. If q_i is a c_2-query, then we charge q_i and m_{i+1} to the query. Also, we perhaps need to pay for mistake m_1. So,

[1]The reason for the care here is a subtlety, which can be illustrated by the following game: imagine performing a random walk on a line, but where an adversary can stop you at any time $t \leq 100$. One might like to say: "if the adversary allows you to make 100 steps, then with high probability you will have made about 50 steps to the right." However, this is false because the adversary's strategy might be to stop the game after your first step to the left (so if you do make 100 steps, at least 99 were to the right). What is true, which can be seen by considering the more powerful adversary that can see your entire sequence of 100 coin tosses in advance, is that with high probability either the adversary stops you before $t = 100$ or by $t = 100$ you will have made about 50 steps to the right.

the number not "paid for" by adversary switches is at most 1 plus twice the number of c_2-queries. Since with high probability we make only $8n^2 \log(2n/\delta)$ c_2-queries, this achieves the bounds claimed. ■

The above two lemmas prove the following theorem.

Theorem 18 *Algorithm* Query-Learn, Stage One, *followed by* Query-Learn, Stage Two *is 1-competitive for learning a pair of adversarially-switched monotone disjunctions.*

5.4 Open Problems

An obvious open problem is to design "competitive" learning algorithms for $k > 2$ switching concepts.

5.5 Related Work

Our switching concepts model is similar to a model considered by Helmbold and Long [38] in which a concept may drift over time, but there are several important differences between our focus and theirs. We restrict the "active concept" to switch between only a small number of concepts instead of drifting through the entire class, but we allow the switches to occur more rapidly and drastically. In particular, just looking at data more recent than some cutoff point need not produce a large set consistent with any individual target.

Experimental work on learning interleaved functions has been done by E. Levin [57]. Ar et al. [10] have examined a similar problem of identifying a set of polynomials over a finite field where each point is assigned a value by one of the polynomials.

Bibliography

[1] H. Abelson and A. diSessa. *Turtle Geometry.* MIT Press, Cambridge, MA, 1980.

[2] S. Akella. Using deterministic actions and incomplete sensors for robot tasks. Thesis proposal, Carnegie Mellon University, July 1994.

[3] S. Albers. The influence of lookahead in competitive paging algorithms. In *First Annual European Symposium on Algorithms (ESA)*, pages 1–12. Springer-Verlag, 1993.

[4] D. Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39:337–350, 1978.

[5] D. Angluin. Inductive inference of formal languages from positive data. *Information and Control*, 45(2):117–135, May 1980.

[6] D. Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51:76–87, 1981.

[7] D. Angluin. Inference of reversible languages. *Journal of the ACM*, 29(3):741–765, July 1982.

[8] D. Angluin. Learning regular sets from queries and counter-examples. Technical Report YALEU/DCS/TR-464, Yale University Department of Computer Science, Mar. 1986.

[9] D. Angluin and L. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. *J. Comput. Syst. Sci.*, 18:155–193, 1979.

[10] S. Ar, R. J. Lipton, R. Rubinfeld, and M. Sudan. Reconstructing algebraic functions from mixed data. In *Proc. of the 33rd Symposium on the Foundations of Comp. Sci.*, pages 503–512. IEEE Computer Society Press, Los Alamitos, CA, 1992.

[11] B. Awerbuch, M. Betke, R. L. Rivest, and M. Singh. BFS without teleportation. Unpublished Manuscript, 1994.

[12] R. Baeza-Yates, J. Culberson, and G. Rawlins. Searching in the plane. *Information and Computation*, 106(2):234–252, 1993.

[13] E. Bar-Eli, P. Berman, A. Fiat, and P. Yan. On-line navigation in a room. In *Proc. 3rd ACM-SIAM SODA*, 1992.

[14] P. Berman and M. Karpinski. Wall problem with convex obstacles. Unpublished Manuscript, July 1994.

[15] M. Betke, R. Rivest, and M. Singh. Piecemeal learning of an unknown environment. In *Proc. 6th ACM Conf. on Computational Learning Theory*, pages 277–286, 1993.

[16] A. Blum and P. Chalasani. Learning switching concepts. In *Proc. 5th Annual Workshop on Computational Learning Theory*, 1992.

[17] A. Blum and P. Chalasani. An online algorithm for improving performance in navigation. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 2–11, 1993.

[18] A. Blum, H. Karloff, Y. Rabani, and M. Saks. A decomposition theory and bounds for randomized server problems. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 197–207, 1992.

[19] A. Blum, P. Raghavan, and B. Schieber. Navigating in unfamiliar geometric terrain. In *Proc. 23rd ACM STOC*, 1991.

[20] M. Blum and D. Kozen. On the power of a compass (or, why mazes are easier to search than graphs). In *Proc. 19th Symp. Foundations of Comp. Sc. (FOCS)*, 1978.

[21] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. In *Proc. 23rd Symp. on Theory of Computing (STOC)*, pages 249–259, 1991.

[22] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 249–259, May 1991. Submitted for publication to Algorithmica's special issue on on-line algorithms.

[23] A. Borodin, N. Linial, and M. Saks. An optimal online algorithm for metrical task systems. In *ACM STOC*, 1987.

[24] P. Chen. Improving path planning with learning. In *Prof. 9th Int'l Workshop on Machine Learning*, 1992.

[25] D. Cheriton and K. Harty. Application-controlled physical memory using external page-cache management. Technical report, Department of Computer Science, Stanford Universit[illegible] 1991. Technical Report draft.

[26] A. D. Christianse *Automatic Acquisition of Task Theories for Robotic Manipulation.* PhD thesis, Carnegie Mellon University, 1992.

[27] F. K. Chung, R. Graham, and M. Saks. A dynamic location problem for graphs. *Combinatorica*, 9(2):111–131, 1989.

[28] T. Darrell and A. Pentland. Classifying hand gestures with a view-based distributed representation. In *Proc. 6th Neural Inf. Proc. Systems*, 1994.

[29] X. Deng, T. Kameda, and C. Papadimitriou. How to learn an unknown environment. In *Proc. 32nd IEEE FOCS*, pages 298–303, 1991.

[30] X. Deng and C. Papadimitriou. E[illegible] an unknown graph. In *Proc. 31st IEEE FOCS*, pages 355–361, 1990.

[31] A. Fiat, D. Foster, H. Karloff, Y. Rabani, Y. Ravid, and S. Vishwanathan. Competitive algorithms for layered graph traversal. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 288–297, 1991.

[32] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. Technical Report CMU-CS-88-196, School of Computer Science, Carnegie Mellon University, 1988. New version appeared in Journal of Algorithms.

[33] E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[34] E. M. Gold. System identification via state characterization. *Automatica*, 8:621–636, 1972.

[35] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37:302–320, 1978.

[36] L. Guibas, R. Motwani, and P. Raghavan. The robot localization problem in two dimensions. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 259–268, 1992.

[37] M. Halldórsson and M. Szegedy. Lower bounds for on-line graph coloring. In *Proc. 3rd Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 211–216, 1992.

[38] D. Helmbold and P. Long. Tracking drifting concepts using random examples. In *Proceedings of the 4th Annual Conference on Computational Learning Theory*, pages 267–280, 1991.

[39] J. Hopcroft, D. Joseph, and S. Whitesides. On the movement of robot arms in 2-dimensional bounded regions. In *Proc. 23rd Symp. Foundations of Comp. Sc. (FOCS)*, 1982.

[40] S. Irani. Coloring inductive graphs online. In *Proc. 31st Annual IEEE Symp. on Foundations of Comp. Sc. (FOCS)*, pages 470–479, 1990.

[41] S. Irani, A. R. Karlin, and S. Phillips. Strongly competitive algorithms for paging with locality of reference. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 228 – 236, Jan. 1992.

[42] B. Kalyanasundaram and K. Pruhs. Visual searching and mapping. Technical report, University of Pittsburgh, 1991.

[43] M.-Y. Kao and S. Tate. Online matching with blocked input. *Inf. Proc. Letters*, 38(1):113–116, 1991.

[44] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator. Competitive snoopy caching. Technical report, Carnegie Mellong University, 1986.

[45] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[46] H. Karloff, Y. Rabani, and Y. Ravid. Lower bounds for randomized k-server and motion-planning algorithms. In *Proc. 23rd ACM STOC*, pages 278–288, 1991.

[47] H. Karloff, Y. Rabani, and Y. Ravid. Lower bounds for randomized k-server and motion-planning algorithms. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 278–288, May 1991.

[48] R. Karp, R. Motwani, and P. Raghavan. Deferred data structuring. *SIAM J. of Computing*, 17(5):883–902, 1988.

[49] M. Kearns and L. G. Valiant. Learning boolean formulae or finite automata is as hard as factoring. Technical Report TR 14-88, Harvard University Aiken Computation Laboratory, 1988.

[50] M. Kearns and L. G. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pages 433–444, Seattle, Washington, May 1989.

[51] T. Kilburn, D. Edwards, M. Lanigan, and F. Sumner. One-level storage system. *IRE Trans. Elect. Computers*, 37:223–235, 1962.

[52] J. Kleinberg. On the localization problem for mobile robots. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, 1994. To appear.

[53] S. Koenig and R.G.Simmons. Complexity analysis of real-time reinforcement learning. In *Proc. AAAI*, pages 99–105, 1993.

[54] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw Hill, 1978.

[55] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Back-propagation applied to handwritten zip code recognition. *Neural Computation*, 1(4), 1989.

[56] K.-F. Lee. *Large-Vocabulary Speaker-independent Continuous Speech Recognition: the Sphinx system.* PhD thesis, Carnegie Mellon University, 1988.

[57] E. Levin. Modeling time variant systems using hidden control neural architecture. In *Proc. Neural Inf. and Proc. Systems 3 (NIPS)*, 1991.

[58] N. Littlestone. Learning when irrelevant attributes abound. In *Proceedings of the Twenty-Eighth Annual Symposium on Foundations of Computer Science*, pages 68–77, Oct. 1987.

[59] T. Lozano-Perez and M. Wesley. An algorithm for planning collision-free paths among polygonal obstacles. *Comm. ACM*, 22:560–570, 1979.

[60] V. Lumelsky. Algorithmic and complexity issues of robot motion in an uncertain environment. *Journal of Complexity*, 3:146–182, 1987.

[61] V. Lumelsky. Algorithmic issues of sensor-based robot motion planning. In *26th IEEE Conference on Decision*, pages 1796–1801, 1987.

[62] V. Lumelsky and A. Stepanov. Dynamic path planning for a mobile automaton with limited information on the environment. *IEEE Trans. on Automatic Control*, 31:1058–1063, 1986.

[63] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11:208–230, 1990. Covers server results from paper in Proceedings of 20th Annual Symposium on Theory of Computing 322–333.

[64] S. McFarling. Cache replacement with dynamic exclusion. In *Proc. 19th Int'l. Symp. on Comp. Arch.*, pages 191–200, 1992.

[65] D. McNamee and K. Armstrong. Extending the mach external pager interface to accomodate user-level page replacement policies. Technical Report 90-09-05, University of Washington Dept. of Comp. Sc. and Engg., 1990.

[66] T. Mitchell, R. Caruana, D. Freitag, J. McDermott, and D. Zabowski. Experience with a learning personal assistant. *Communications of the ACM*, July 1994.

[67] K. S. Narendra and M. A. L. Thathachar. Learning automata – a survey. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-4(4):323–334, 1974.

[68] B. Oomen, S. Iyengar, N. Rao, and R. Kashyap. Robot navigation in unknown terrains using learned visibility graphs. *IEEE Journal of Robotics and Automation.*, 1987.

[69] M. Palmer and S. Zdonik. Fido: A cache that learns to fetch. In *Proc. 1991 Int'l Conf. on Very Large Databases*, 1991.

[70] C. Papadimitriou and M. Yannakakis. Shortest paths without a map. In *Proc. 16th ICALP*, 1989.

[71] D. A. Pomerleau. *Neural Network Perception for Mobile Robot Guidance.* PhD thesis, Carnegie Mellon University, 1992.

[72] N. Rao, S. Iyengar, and G. deSaussure. The visit problem: visibility graph based solution. In *IEEE Int'l. conf. Rob. and Automation*, 1988.

[73] J. Reif. Complexity of the mover's problem and generalizations. In *Proc. 20th Symp. Foundations of Comp. Sc. (FOCS)*, 1979.

[74] R. L. Rivest and R. E. Schapire. Diversity-based inference of finite automata. In *Proceeding of the Twenty-Eighth Annual Symposium on Foundations of Computer Science*, pages 78–87, Los Angeles, California, Oct. 1987.

[75] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pages 411–420, Seattle, Washington, May 1989.

[76] K. Salem. Adaptive prefetching for disk buffers. Technical Report TR-91-64, Goddard Space Flight Center, 1991.

[77] J. Schwartz and M. Sharir. On the piano movers problem: I. the case of a two-dimensional rigid polygonal body moving amidst polygonal barriers. *Comm. Pure Appl. Math.*, 33:345–398, 1983.

[78] J. T. Schwartz and M. Sharir. On the piano movers problem: Ii. general techniques for computing tological properties of real algebraic manifolds. *Adv. in Appl. Math.*, 4:298–351, 1983.

[79] D. D. Sleator and R. T. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, Feb. 1985.

[80] S. Thrun. Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, Carnegie Mellon University, 1992.

[81] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, Nov. 1984.

[82] J. S. Vitter and P. Krishnan. Optimal prefetching via data compression. In *Proc. 32nd Symp. on Foundations of Comp. Sc. (FOCS)*, pages 121–130, 1991.

[83] A.-C. Yao. Probabilistic computations: Toward a unified measure of complexity. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 222–227, 1977.

[84] N. Young. *Competitive Paging and Dual-Guided On-Line Weighted Caching and Matching Algorithms.* PhD thesis, Princeton University, 1991. Available as Computer Science Dept. Tech. Report CS-TR-348-91.

[85] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory*, 24:530–536, 1978.

Index

JumpDownLeft, 40
    general, 53
JumpDownRight, 43
    general, 51
τ-fence, 47
τ-post, 47

access graph, 75
automata
    inferring, 61

band, 22
BRS algorithm, 22

cell, 68
    -tree, 70
    big, 69
    small, 69
    span of, 69
competitive ratio, 2
competitiveness, 2

deferred data structuring, 64
demos, 29–33
doubling strategy, 28
drifting concepts, 86

exploration, 60
    -exploitation tradeoff, 62
    piecemeal, 61

fence, 22
    -tree, 29
      -rules, 35
      traversal, 37
    crossing, 22

function-approximation, 1

homing sequences, 63

invariant
    Almost-Ordering, 40
    Jump-Up, 44
    Last-Node, 38
    Ordering, 40
    Progress-Ordering, 38

locality of reference, 75
localization, 63
lookahead, 3, 74
lower bound
    for k-trip navigation, 18
    for sleepy pager, 66
    for switching concepts, 80

membership queries, 5
mistake bound, 5

online algorithms, 2
online learning, 5
    of switching concepts, 80

partial information, 3, 16
path-planning
    with arbitrary obstacles, 62
    with complete information, 63
performance-improvement, 3, 9
    in navigation, 15
    in paging, 65
phase, 68
piano movers problem, 63
prefetching, 76

pursuit-evasion game, 67

reinforcement learning, 62

scene, 16
    simple, 21
search-quality tradeoff, 27
sleepy pager, 66
sweep, 22

task system, 9
    hidden, 10
      navigation as, 57
      paging as, 73
      reversible, 11

watchman's route, 60

Glossary

For quick reference we summarize below some notation and terms used in the thesis.

Hidden Task Systems

S	Set of states in a task system.
s	Equals $\|S\|$, the number of states in S.
d	State transition cost matrix: $d(i,j)$ = cost of moving from state i to state j.
$\mathbf{T}$	Sequence of tasks to be processed (perhaps repeatedly).
T^i	The ith task in the sequence $\mathbf{T}$.
A	The online scheduling algorithm.
$c_A(\mathbf{T})$	The cost (sum of state transition costs and task processing costs) of processing $\mathbf{T}$ with algorithm A.
$c_0(\mathbf{T})$	The optimal offline cost of processing $\mathbf{T}$, with complete knowledge of processing costs of all tasks of $\mathbf{T}$ in every state of S.
t	The number of repetitions of a task sequence.
$\rho(A,m,s,t)$	The average competitive ratio of algorithm A after processing a sequence of m tasks t times, in a hidden task system with s states.
$\rho_i(A,m,s)$	The "instantaneous" competitive ratio of algorithm A on the ith repetition of a sequence of m tasks, in a hidden task system with s states.

Robot Navigation

s	Start point, can be assumed to be $(0,0)$.
t	Target. Usually the line $x = n$.
(x,y)	Current robot coordinates at any time.
L	Length of shortest s-t path that avoids obstacles.
n	Euclidean s-t distance.
k	Number of trips.
$R_i(S)$	Distance traveled by robot using strategy R on the ith s-t trip in scene S.
$R^{(k)}(S)$	Total distance traveled by R in scene S during the first k trips.
$\rho(R,n,k)$	Average competitive ratio of algorithm R at the at the end of k trips, in a scene where the Euclidean s-t distance is n.
$\rho_i(R,n)$	"Instantaneous" competitive ratio on the ith trip.
h	Half the obstacle-height in a simple scene.
τ	Equals $L/\sqrt{nk}$.
a	In a simple scene, equals h/τ.
G	Number of fences per group, equals $\lceil \frac{k}{a} \rceil$ in a simple scene and equals k in a general scene.
M	Number of obstacles per fence in the group, equals $\lceil \frac{k}{a} \rceil + \lceil \frac{2L}{h} \rceil$ in a scene, and $k + \lceil \frac{2L}{\tau} \rceil$ in a general scene.
F_i	The ith (up) fence in a group, where the *highest/leftmost* fence is F_1.
$F_i(m)$	The mth obstacle/τ-post in F_i, where the leftmost obstacle/post of F_i is $F_i(1)$.
$(X_i(m), Y_i(m))$	(x,y)-coordinates of the mth obstacle/τ-post in the ith fence.
M_i	The number of obstacles/posts in fence F_i so far.
X_i	The x-coordinate of the rightmost obstacle/post of fence F_i.

Simple Scene A scene where all obstacles have height $2h$ and width 1, and the lower left hand corners of the obstacles are at coordinates of the form (i, jh) for integer i and j.

Search Trip A trip that walks a distance $O(L\sqrt{nk})$ and finds a path of length $O(L\sqrt{n/k})$.

Window Rectangular region of height $2L$ between s and t, centered vertically at s.

Up Fence In a simple scene, a sequence of obstacles extending across the window, where each obstacle is exactly h higher than and some positive amount to the right of the preceding obstacle in the sequence. In a general scene, replace "obstacle" by "τ-post", and τ by h.

τ-post A portion of the left side of an obstacle, of height 2τ.

Progress-Ordering Invariant. At any time during the construction of the fence-tree by our algorithm, no fence has more obstacles/posts than a fence above it.

Last-Node Invariant. At any time, a new edge is only added from the *last* (highest) node of a partial fence.

Ordering Invariant. All fences below the current fence F_i are x-ordered, i.e., for all $j > i$, $X_j \leq X_{j+1}$.

Almost-Ordering Invariant. No fence has more than one obstacle/post to the right of a lower fence, i.e., for $i < j$, $X_i(M_i - 1) \leq X_j$.

Jump-Up Invariant. Whenever the algorithm jumps from a fence F_i to the fence F_{i-1} above, either F_i has the same number of obstacles as F_{i-1}, or it has exactly one less obstacle and its last obstacle is left of the last obstacle of F_{i-1}, i.e. $X_i < X_{i-1}$. For general scenes, read "post" wherever "obstacle" appears.

Paging

n Total number of pages addressed.

k Cache size.

t Number of repetitions of a fixed page request sequence.

Sleepy Pager A pager that is only woken up when there is a page fault, and is unaware of page requests that occur before it is next woken up.

$\rho(P, n, k, t)$ Average competitive ratio of paging algorithm P after t repetitions of a fixed page request sequence, in an n-page, cache-size k paging system.

$\rho(P, k, t)$ Same as $\rho(P, n, k, t)$, but for $n = k + 1$.

σ Either the actual request sequence being repeated, or the sequence of known requests so far.

$P^{(t)}(\sigma)$ Total number of page faults of algorithm P during the first t iterations of σ.

$\sigma(i)$ ith (known) request.

Cell $[i, j)$ The time period starting just before request $\sigma(i)$ and ending just before request $\sigma(j)$.

Span For a cell, the number of first-iteration page faults in the cell.

Big Cell A cell whose span is greater than k/t.

Small Cell A cell that is not big.

Phase A cell where every page is known to be requested at least once.

Switching Concepts

c_1, c_2 The two switching concepts.

c Current concept, either c_1 or c_2.

n Number of variables (in a monotone disjunction).

s Number of switches so far.

$R(c_i)$ The set of relevant variables of c_i.